On Efficient Polynomial Multiplication and Its Impact on Curve based Cryptosystems

by

Ahmad Salam Alrefai

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Secure communication is critical to many applications. To this end, various security goals can be achieved using elliptic/hyperelliptic curve and pairing based cryptography. Polynomial multiplication is used in the underlying operations of these protocols. Therefore, as part of this thesis different recursive algorithms are studied; these algorithms include Karatsuba, Toom, and Bernstein. In this thesis, we investigate algorithms and implementation techniques to improve the performance of the cryptographic protocols. Common factors present in explicit formulæ in elliptic curves operations are utilized such that two multiplications are replaced by a single multiplication in a higher field. Moreover, we utilize the idea based on common factor used in elliptic curves and generate new explicit formulæ for hyperelliptic curves and pairing. In the case of hyperelliptic curves, the common factor method is applied to the fastest known even characteristic hyperelliptic curve operations, i.e. divisor addition and divisor doubling. Similarly, in pairing we observe the presence of common factors inside the Miller loop of Eta pairing and the theoretical results show significant improvement when applying the idea based on common factor method. This has a great advantage for applications that require higher speed.

Acknowledgements

I would like to express my deep gratitude to my supervisor Professor Anwar Hasan for his constructive feedback, unconditional help, and continuous support throughout the thesis work. I would also like to thank the readers of my thesis Professors Gordon Agnew and Guang Gong for their great assistance and useful comments. I am grateful to Professor David Jao for his valuable input. I would like to acknowledge Dr. Murat Cenk with whom I had useful discussions that have a great impact on this work. In addition, I appreciate helpful email correspondence with Dr. Jithra Adikari and Dr. Christophe Negre. I am also thankful to Susan Gow and John Vardon for their editing of parts of the thesis.

Dedication

I dedicate this thesis to my parents Abdullateef and Wiam and to my siblings Muhammad, Muzna and Nour.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Symbols

AES advanced encryption standard

B3W Bernstein 3-way split

DES data encryption standard

ECC elliptic curve cryptography

FE3W field extension based 3-way split

HEC hyperelliptic curves

IK3W improved Karatsuba 3-way split

K2W Karatsuba 2-way split

K3W Karatsuba 3-way split

NAF non-adjacent form

Chapter 1

Introduction

This chapter briefly introduces the work of this thesis for which the main aim is to improve and efficiently implement certain algorithms for cryptographic systems. These include elliptic, hyperelliptic curves and pairing based cryptography. In addition, algorithms related to polynomial multiplication and other finite field operations are presented because they are considered basic blocks of group operations over elliptic curves.

Section 1.1 of this chapter gives a brief overview of the basic communication model and public key cryptography; it illustrates the use of cryptographic protocols for communication. In Section 1.2, we show an introduction to private key cryptography and public key cryptography and also we explain how the elliptic curve cryptographic schemes are examples of public key cryptography. A short description of the motivation to conduct this study and the objectives are then highlighted. The final section presents the proposal organization.

1.1 Communication Model

Assume that we have two communicating parties Alice (A) and Bob (B), who want to communicate securely using an insecure communication channel where Eve (E), who is a malicious adversary, is trying to listen to the channel. Cryptography is about designing mathematical and algorithmic techniques so that Alice and Bob can communicate securely

and Eve cannot learn the original message, impersonate either parties, or modify the message. Figure 1.1 shows these parties in what is called basic communication model.

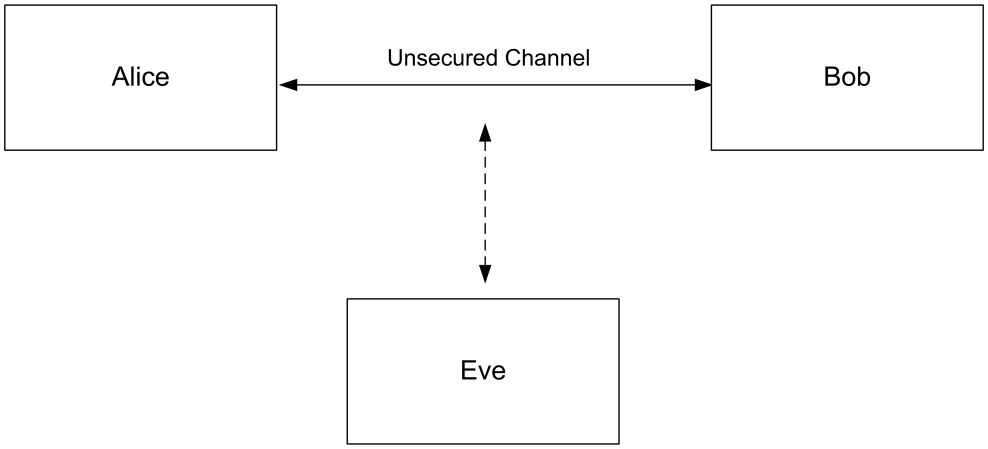


Figure 1.1: Basic communication model.

Basically, there are five major *security goals* of secure communication, namely, confidentiality, data integrity, data origin authentication, entity authentication, and non-repudiation. *Confidentiality* is when unauthorized users cannot read the message, i.e., when A sends a message to B, E cannot understand its content. *Data integrity* means that the system should provide capability to make sure that the message has not been modified. For example, if E modifies the message sent from A, B can detect this. *Data origin authentication* is the ability to confirm the data source. When the message claimed to be from A is received by B, the latter should be able to verify that it is really sent from A. *Entity authentication* is to confirm the identity of an entity. For example, B should be able to trust that the identity of the other communicating party is A. *Non-repudiation* means that previous actions or commitments cannot be denied by an entity. If A sent a message to B, then B can convince a neutral third party that it is indeed from A, and A cannot deny that.

1.2 Symmetric Key Cryptography vs. Asymmetric Key Cryptography

There are two types of cryptographic systems: symmetric key cryptography and assymmetric key cryptography. In *symmetric key cryptography*, shown in Figure 1.2a, both parties agree on a secret and authentic keying material (channel). They can use symmetric key encryption, like data encryption standard (DES) or advanced encryption standard (AES), to provide confidentiality. The communicating parties can also use a message authentical code, such as (HMAC), to provide data integrity and data origin authentication.

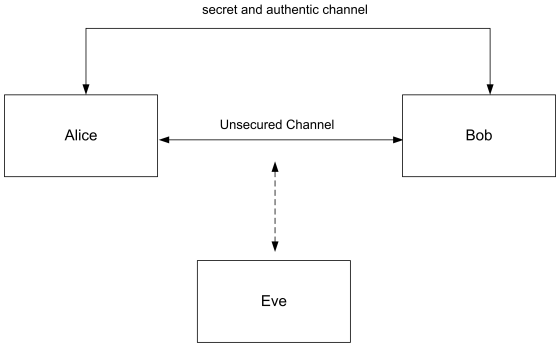
The symmetric key algorithms are known to be efficient. However, there are two major disadvantages of symmetric key cryptography. The first one is the *key distribution problem*. Since both parties have to agree on a key and the distribution must be done in a secret and authentic way, the existence of a secure channel is assumed. The second problem is the *key management problem*. If we have a network of N nodes, in this case each node must maintain $N - 1$ secret keys.

Diffie and Hellman [1] introduced the assymetric key cryptography system in order to solve the key distribution problem without the need of a secure channel. This type is shown in Figure 1.2b. Each party selects a set of keying material (e, d) where e is public key and d is private key, and it is computationally infeasible to know the private key given the public key.

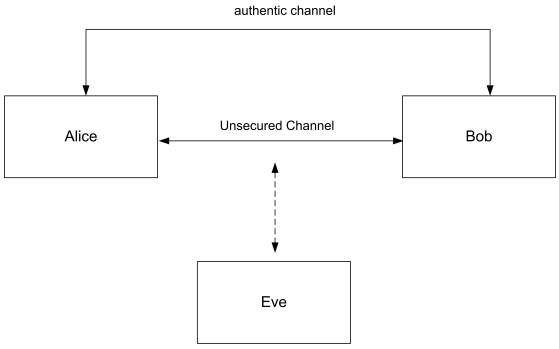
Asymmetric key cryptography depends on the intractability of a number-theoretic problem. RSA asymmetric key encryption and signature schemes [2] are based on the integer factorization problem. Elgamal asymmetric key encryption and signature schemes [3] are based on the discrete logarithm problem. Finally, elliptic curve cryptographic schemes are based on the elliptic curve discrete logarithm problem.

1.3 Motivation

Many cryptographic applications are based on elliptic and hyperelliptic curve cryptography. Examples include ciphering, deciphering, signing, and verifying a message. Digital signature can be done in a way opposite to encryption and decryption, in RSA for example. The private key d can be used to sign the message $S = sign(M, d)$; then the



(a) Symmetric key cryptography.



(b) Public key cryptography.

Figure 1.2: Symmetric key vs. public key cryptography.

other party can use the public key e of the sender to verify the signature of the message $d = verify(S, e, M)$.

Pairing based cryptography has many cryptographic applications, like the three party key agreement protocol, identity-based cryptography, and short signatures [4]. The two party key agreement protocol is based on the Diffie-Hellman algorithm, which is a one round protocol because of the independence of each message [1]. Figure 1.3 shows how to conduct the three-party two-way key agreement protocol in two-round. The secret key is $abcP$ and it is intractable given P, aP, bP, cP, abP, bcP, and caP. Pairing can be used to conduct the three party key agreement protocol in one round as shown in Figure 1.4.

Hyperelliptic curves have applications in asymmetric key cryptography, design of error correcting codes, and integer factorization algorithms [5]. An elliptic curve is a special case of a hyperelliptic curve. Compared to elliptic curves, hyperelliptic curves have remained

less explored, especially for efficient implementation. Therefore, studying how to improve the algorithms in hyperelliptic curves is of our interest.

In conclusion, improving the basic algorithms in these cryptographic fields and efficiently implementing them are critical to the applications that use them. Since we are considering software implementation, looking into methods to improve performance is very helpful.

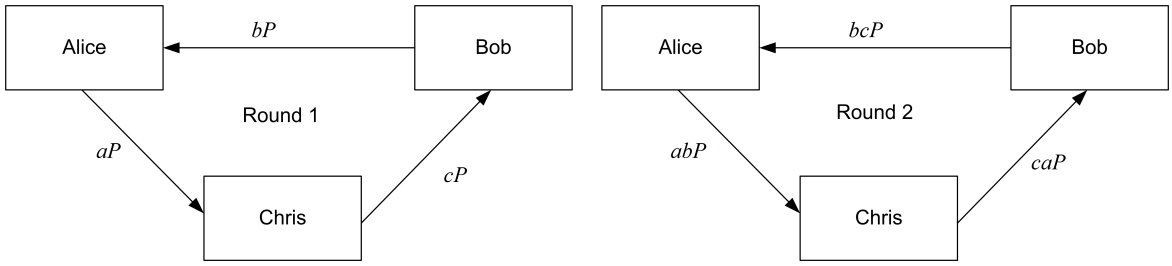


Figure 1.3: Three-party two-round key agreement protocol.

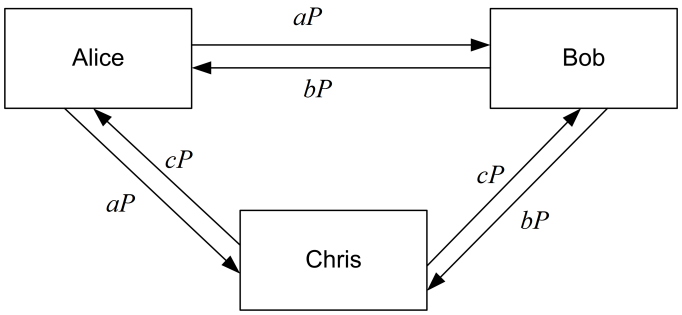


Figure 1.4: Three-party one-round key agreement protocol.

1.4 Objectives

Elliptic curve cryptography has a shorter key length than RSA for the same security level. Point addition and point doubling are the major operations of elliptic curve cryptography. In [6], a method has been proposed to improve algorithms of point addition and point doubling. Both algorithms use polynomial multiplication as a major operation. In

[7], several polynomial multiplication algorithms have been investigated. Pairing and hyperelliptic curve based cryptography has not received as much attention as elliptic curve cryptography.

In this study, we will extend the idea presented in [6] and use it in pairing computations and hyperelliptic curves operations to improve their performance. We will investigate different techniques to improve polynomial multiplication and scalar multiplication in hyperelliptic curves.

1.5 Organization

In Chapter 2, the mathematical background needed to understand the proposal is presented. This chapter introduces abstract algebra, finite field arithmetic, elliptic curve cryptography, pairing based cryptography, and hyperelliptic curves cryptography. Then, Chapter 3 presents some mathematical analyses of the methods to improve point addition and point doubling algorithms. Chapters 4 and 5 explain in detail the software implementation and the timing results respectively. In Chapter 6, improving the performance of pairing and hyperelliptic curves is investigated. Finally, a summary of our present work and possible directions of future research are presented in Chapter 7.

Chapter 2

Mathematical Background

In this chapter we introduce the mathematical foundation needed to understand the rest of the proposal. Specifically, elliptic curve cryptography (ECC) is taken into consideration. Refer to [8] for more detailed explanation about ECC.

This chapter is organized as follows. First, we give a brief introduction to some abstract algebra concepts. We then present finite field arithmetic and review some operations like addition, multiplication, squaring, reduction, and inversion. After that, we focus on elliptic curve cryptography and look at its definition, point representation and its operations like point addition, point doubling and scalar multiplication. Next, we give brief introduction to pairing based cryptography. Finally, we give a brief overview of hyperelliptic curves.

2.1 Abstract Algebra

In this section, we look at concepts from abstract algebra necessary to understand ECC. Specifically, we review groups and fields.

2.1.1 Groups

An abelian group $(G, *)$ is a set G with a binary operation $* : G \times G \rightarrow G$ satisfying the following properties

- (*Associativity*) $a * (b * c) = (a * b) * c$ for all $a, b, c \in G$.
- (*Existence of an identity*) $e \in G$ and $a * e = e * a = a$ for all $a \in G$.
- (*Existence of inverses*) for each $a \in G$, there exists $b \in G$, $a * b = b * a = e$.
- (*Commutatively*) $a * b = b * a$ for all $a, b \in G$.

The group might be either an additive group and in this case the negative of a is denoted as $-a$ or a multiplicative group and the inverse of a is denoted as a^{-1}. The group is finite if G is a finite set, the number of elements is called the *order* of G.

Let p be a prime and let e denote the set of integers modulo p, then $(\mathbb{F}_p, +)$ is a finite additive group of order p and its additive identity element is 0. Let $\mathbb{F}_p^*$ denote the nonzero set of elements in $\mathbb{F}_p$, then $(\mathbb{F}_p^*, \cdot)$ is a finite multiplicative group of order $p - 1$ with multiplicative identity of 1. If G is a finite multiplicative group of order n and $g \in G$, the smallest positive integer t such that $g^t = 1$ is called the *order* of g; such a t always exists and it divides n. The set $\langle g \rangle = g^i : 0 \leq i \leq t - 1$ of all powers of g is a cyclic subgroup of G generated by g. In the case additive groups, it is similar but $tg = 0$ and $\langle g \rangle = ig : 0 \leq i \leq t - 1$. If G has an element g of order n, then G is said to be *cyclic* group and g is the *generator* of G.

2.1.2 Fields

Fields are abstractions of similar other number systems like rational ($\mathbb{Q}$), real ($\mathbb{R}$), or complex ($\mathbb{C}$) number systems and their basic properties. They are composed of a set $\mathbb{F}$ with two operations: addition ($+$) and multiplication ($\cdot$) such that the following properties are satisfied:

- $(\mathbb{F}, +)$ is an abelian group with an additive identity (0).
- $(\mathbb{F} \backslash 0, \cdot)$ is an abelian group with a multiplicative identity (1).
- The distributive law holds: $(a + b) \cdot c = a \cdot c + b \cdot c$ for all $a, b, c \in \mathbb{F}$.

The field is said to be finite if $\mathbb{F}$ is finite. So, the triple $(\mathbb{F}_p, +, \cdot)$ is a finite field denoted as $\mathbb{F}_p$. Subtraction can be defined in terms of addition. Assuming $a, b \in \mathbb{F}$, then $a - b = a + (-b)$, where $-b$ is the negative of b such that $b + (-b) = 0$. Similarly, division can be defined in terms of multiplication, so $a/b = a \cdot b^{-1}$ where b^{-1} is the inverse of b such that $b \cdot b^{-1} = 1$.

Let p be a prime number. If addition and multiplication are performed over integers modulo p, then it is a finite field of order p denoted by $\mathbb{F}_p$. Reduction modulo p for any integer a can be done by taking $a \mod p$ and the result is the remainder r where $0 \leq r \leq p - 1$ obtained upon dividing a by p.

If the order is of the form 2^m, then the field is binary. Polynomials can be used to represent the field where coefficients are in the field $\mathbb{F}_2$ and the degree is at most $m - 1$.

$$\mathbb{F}_{2^m} = a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \ldots + a_2z^2 + a_1z + a_0 : a_i \in \{0, 1\}$$

Addition of two field elements is done by addition modulo 2 of the corresponding coefficients. Field multiplication is done by polynomial multiplication modulo an *irreducible* polynomial of degree m. The latter cannot be represented as a product of other binary polynomials each of degree less than m.

When the order is p^m, where p is a prime and $m \geq 2$, the following equation generalizes the polynomial basis representation to the field $\mathbb{F}_{p^m}$.

$$\mathbb{F}_{p^m} = a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \ldots + a_2z^2 + a_1z + a_0 : a_i \in \mathbb{F}_p$$

We call k to be a subfield of a field K if k is a field with the operators of K and if k is a subset of K. Also, K is considered an extension field of k. A finite field $\mathbb{F}_{p^m}$ has one subfield for every divisor l of m; its elements are the elements $a \in \mathbb{F}_{p^m}$ such that $a^{p^l} = a$. Arithmetic operations can be done to each type. These includes addition, multiplication, squaring, reduction and inversion. We will look at these operations in the binary field since it is the focus of the proposal.

2.2 Binary Field Arithmetic

This section focuses on the binary field algorithms and their implementation. These algorithms include addition, multiplication, squaring, reduction and inversion. Before

describing the algorithms, we need to look at the representation of the field in software. Since the field is binary, each bit in the computer system can represent a polynomial coefficient. Assume that the computer is W-bit architecture, and W is a multiple of 8. Assume also that we have a field $\mathbb{F}_{2^m}$, and consider an element in this field $a(z) = a_{m-1}z^{m-1} + \ldots + a_2z^2 + a_1z + a_0$. To represent this element we need just the vector $a = (a_{m-1}, \ldots, a_2, a_1, a_0)$; this vector has a length of m bits. Then, we can store a in software in an array of t W-bit words, where $t = \lceil m/W \rceil$, in the following way: $A = (A[t-1], \ldots, A[2], A[1], A[0])$. A field element representation is shown in Figure 2.1. The leftmost s bits of $A[t-1]$ are unused bits, where $s = Wt - m$. The right most bit of $A[0]$ is a_0.

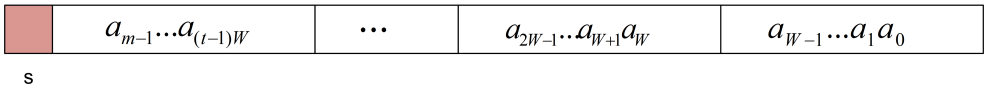


Figure 2.1: Representation of $a \in \mathbb{F}_{2^m}$.

Before going into details, we need to look at the notations used in the algorithms. Assume that we have two W-bit words, U and V; then the notations used to denote the operations are the following:

$U \oplus V$: bitwise exclusive-or.

$U\&V$: bitwise AND.

$U \ll i$: left shift of U by i positions.

$U \gg i$: right shift of U by i positions.

2.2.1 Addition

Since adding elements in binary fields is done modulo 2, then a bitwise XOR can simply be used to add two polynomials. Algorithm 2.1 shows how we can add two polynomials in software.

Algorithm 2.1 Addition in $\mathbb{F}_{2^m}$.

1: **procedure** PolynomialAdd$(a(z), b(z))$ ▷ $c(z) = a(z) + b(z)$.
2: **for** i from 0 to $t - 1$ **do**
3: $c[i] \leftarrow A[i] \oplus B[i]$
4: **end for**
5: **return** (c).
6: **end procedure**

2.2.2 Multiplication

Multiplication can be done in two steps. The first one is to multiply the inputs as polynomials and the second one is to reduce the results modulo $f(z)$. In this section we describe the multiplication operation. Algorithm 2.2 shows how to multiply two polynomials by the use of the right-to-left comb method. Note that the notation $C\{j\}$ is used to represent the truncation of the upper words starting from word j; so if $C = (C[n], \ldots, c[2], c[1], c[0])$ then $C\{j\} = (C[n], \ldots, C[j+1], C[j])$. The inner loop deals with the words and the outer loop deals with the bit position. In each iteration of the outer loop, i.e. bit position, the value of B is shifted left by 1 position (multiplied by z).

2.2.3 Squaring

If we have a polynomial $a(z) = a_{m-1}z^{m-1} + ... + a_2z^2 + a_1z + a_0$ then

$$a(z)^2 = a_{m-1}z^{2m-2} + ... + a_2z^4 + a_1z^2 + a_0$$

So squaring the polynomial results in a polynomial of double the size of the original polynomial. The coefficients is one when the power of z is even or 0. Hence, the coefficients where the power is odd are zeros. Therefore, we can compute the square of a polynomial by inserting zeros between every two binary digits as shown in Figure 2.2.

In order to make the squaring faster, one can use a lookup table that converts an 8-bit polynomial into the corresponding 16-bit polynomial which represents the square of the input. Algorithm 2.3 shows the polynomial squaring algorithm.

Algorithm 2.2 Right-to-left comb method for polynomial multiplication.

procedure PolynomialMultiply($a(z), b(z)$) ▷ $c(z) = a(z) \cdot b(z)$
    $C \leftarrow 0$.
    for k from 0 to $W - 1$ **do**
        for j from 0 to $t - 1$ **do**
            if the kth bit of $A[j]$ is 1 **then**
                add B to $C\{j\}$
            end if
        end for
        if $k \neq (W - 1)$ **then**
            $B \leftarrow B \cdot z$.
        end if
    end for
    return (c).
end procedure

Algorithm 2.3 Polynomial Squaring with $w = 32$.

procedure PolynomialSquaring($a(z)$) ▷ $c(z) = a(z)^2$
    Precomputation. For each byte $d = (d_7, \ldots, d_1, d_0)$, compute the 16-bit quantity $T(d) = (0, d_7, \ldots, 0, d_1, 0, d_0)$.
    for i from 0 to $t - 1$ **do**
        Let $A[i] = (u_3, u_2, u_1, u_0)$ where each u_j is a byte ▷ assuming W is 32 bits.
        $C[2i] \leftarrow (T(u_1), T(u_0)), C[2i + 1] \leftarrow (T(u_3), T(u_2))$ ▷ look up table of 64 entries.
    end for
    return (c)
end procedure

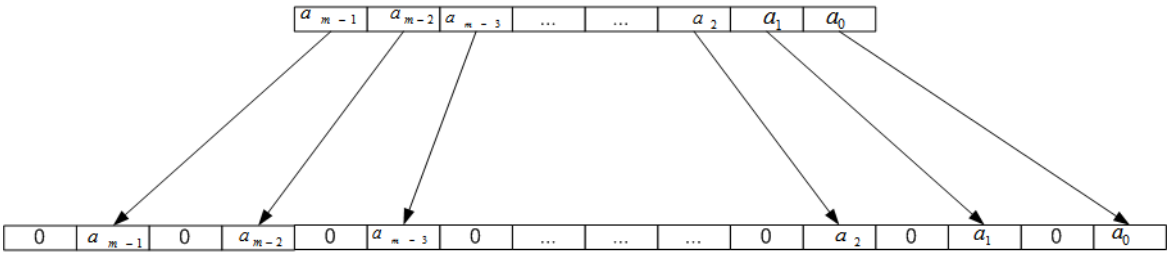


Figure 2.2: Squaring a binary polynomial $a(z) = a_{m-1}z^{m-1} + ... + a_2z^2 + a_1z + a_0$.

2.2.4 Reduction

After performing polynomial multiplication or squaring, the resultant polynomial is one of degree at most $2m-2$. Therefore, a reduction is needed such that the result is an element in the field of degree at most $m-1$. The reduction polynomial $f(z)$ is of degree m.

We can write $f(z) = z^m + r(z)$ where $r(z)$ is a binary polynomial of degree at most $m-1$. In order to reduce $c(z)$ of degree $2m-2$, the following observation is used:

$$\begin{aligned} c(z) &= c_{2m-2}z^{2m-2} + \cdots + c_mz^m + c_{m-1}z^{m-1} + \cdots + c_1z + c_0 \\ &\equiv (c_{2m-2}z^{m-2} + \ldots + c_m)r(z) + c_{m-1}z_{m-1} + \ldots + c_1z + c_0 \end{aligned}$$

This is because $r(z) \equiv z^m$ where we equate the reduction polynomial to 0. Then z^m can be taken as a common factor and replaced by $r(z)$. Algorithm 2.4 shows how to perform modular reduction one bit at a time.

Faster reduction can be achieved if NIST reduction polynomials are used. These are either trinomial or pentanomial. So in order to reduce the bits with order greater than $m-1$, they need to be added (XORed) a number of times to the polynomial with proper shifts. Assume we need to reduce the NIST polynomial modulo $f(z) = z^{233} + z^{74} + 1$; so $m = 233$. Also assume that $W = 32$; hence $t = 8$. The word C[10] represents the polynomial $c_{351}z^{351} + \ldots + c_{321}z^{321} + c_{320}z^{320}$. It can be written in terms of lower degrees as follows:

Algorithm 2.4 Modular reduction (one bit at a time).

procedure MODULAREREDUCTION($c(z), f(z)$) ▷ $c(z)$ is a binary polynomial of degree at most $2m-2$
▷ $f(z) = z^m + r(z)$
  Precomputation. Compute $u_k(z) = z^k r(z), 0 \leq k \leq W-1$.
  for i from $2m-2$ downto m **do**
    if $c_i = 1$ **then**
      Let $j \leftarrow \lfloor (i-m)/W \rfloor$ and $k \leftarrow (i-m) - Wj$.
      $C\{j\} \leftarrow C\{j\} + u_k(z)$.
    end if
  end for
  return $(C[t-1], \ldots, C[1], C[0])$.
end procedure

$$
\begin{aligned}
z^{320} &\equiv z^{161} + z^{87} \mod f(z) \\
z^{321} &\equiv z^{162} + z^{88} \mod f(z) \\
&\vdots \\
z^{351} &\equiv z^{192} + z^{118} \mod f(z)
\end{aligned}
$$

The above congruences has two columns in the right hand side. In order to reduce $C[10]$, one can add (XOR) it two times with its rightmost bit added to bits 161 and 87. Figure 2.3 shows how this can be performed.

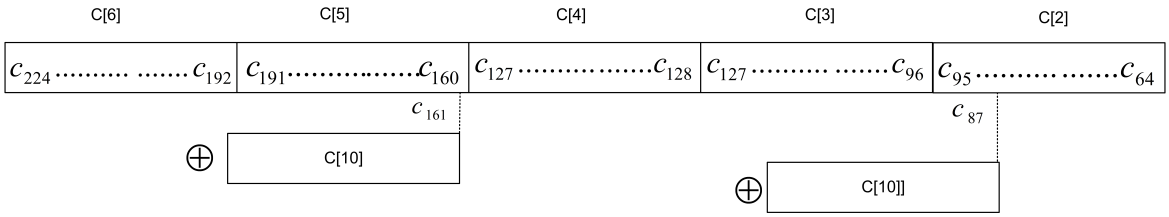


Figure 2.3: Reducing the 32-bit word $C[10]$ modulo $f(z) = z^{233} + z^{74} + 1$.

This can work for all reduction polynomials recommended by NIST in [9]:

$$\begin{aligned} f(z) &= z^{163} + z^7 + z^3 + 1 \\ f(z) &= z^{233} + z^{74} + 1 \\ f(z) &= z^{283} + z^{12} + z^7 + z^5 + 1 \\ f(z) &= z^{409} + z^{87} + 1 \\ f(z) &= z^{571} + z^{10} + 1. \end{aligned}$$

2.2.5 Inversion

Inversion can be done based on the Euclidean algorithm for polynomials. Assuming that a is a binary polynomial, the inverse of an element $a \in \mathbb{F}_{2^m}$ is $g \in \mathbb{F}_{2^m}$ such that $ag \equiv 1 \mod f$. Assume that b is also a binary polynomial, then $\gcd(a, b) = \gcd(b - ca, a)$ for all binary polynomials c. Assume that f is an irreducible reduction polynomial of degree m and a is a binary polynomial of degree $m - 1$, therefore $\gcd(a, f) = 1$. If we can write a polynomial $ag + fh = 1$ where g and h are binary polynomials. Then $ag = 1$ and $g = a^{-1}$. Algorithm 2.5 shows the inversion algorithm.

Algorithm 2.5 Extended Euclidean algorithm based inversion in $\mathbb{F}_{2^m}$.

1: **procedure** PolynomialInverse(a) ▷ a is nonzero polynomial of degree at most $m - 1$.
2: $u \leftarrow a, v \leftarrow f$.
3: $g_1 \leftarrow 1, g_2 \leftarrow 0$.
4: **while** $u \neq 1$ **do**
5: $j \leftarrow \deg(u) - \deg(v)$.
6: **if** $j < 0$ **then**
7: $u \leftrightarrow v, g_1 \leftrightarrow g_2, j \leftarrow -j$.
8: **end if**
9: $u \leftarrow u + z^j v$.
10: $g_1 \leftarrow g_1 + z^j g_2$.
11: **end while**
12: **return** g_1.
13: **end procedure**

2.3 Elliptic Curve Cryptography

Koblitz [10] and Miller [11] independently proposed the use of elliptic curves over finite fields to design cryptographic schemes. This section covers the basics of ECC which include its definition, point representation, and its basic algorithms for point doubling, point addition and point multiplication. We finally look at the use of NAF representation and look at the window based NAF point multiplication.

2.3.1 Definition

An elliptic curve E over a field K is defined by an equation called *Weierstrass* equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

The coefficients of E must be in the field $a_1, a_2, a_3, a_4, a_6 \in K$ and the *discriminant* of E must be nonzero $\Delta \neq 0$. Discriminant Δ is defined as $\Delta = -{d_2}^2 d_8 - 8{d_4}^3 - 27{d_6}^2 + 9d_2d_4d_6$, where $d_2 = {a_1}^2 + 4a_2$, $d_4 = 2a_4 + a_1a_3$, $d_6 = {a_3}^2 + 4a_6$, and $d_8 = {a_1}^2a_6 + 4a_2a_6 - a_1{a_3}^2a_4 + a_2{a_3}^2 - {a_4}^2$. If L is an extension field of K, then the set of L-*rational points* on E is:

$$E(L) = \{(x, y) \in L \times L : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{\infty\}$$

where ∞ is the *point at infinity*.

2.3.2 ECC vs. RSA

Table 2.1 shows the key size of Elliptic curve and RSA cryptosystems for equivalent security levels. Generally, ECC requires shorter key sizes. As the security level goes higher the difference of key size between ECC and RSA becomes more significant. This shows the importance of using elliptic curve cryptography, especially for mobile applications and embedded systems.

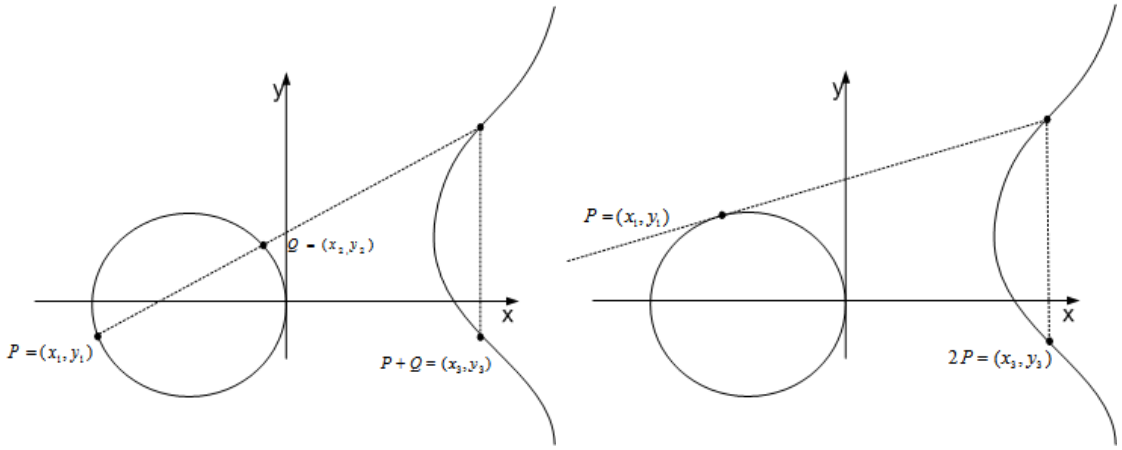
2.3.3 Group law

The *chord-and-tangent* rule for point addition and point doubling in $E(K)$ is shown in Figure 2.4. The set of points in $E(K)$ with (∞) forms an abelian group. The addition rule

Table 2.1: Key sizes for ECC and RSA for equivalent security levels.

Security Level	80	112	128	192	256
ECC	160	224	256	384	512
RSA	1024	2048	3072	8192	15360

of two distinct points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on the curve can be done by drawing a line from P to V and this line will intersects the curve at a third point. The reflection of this point about the x-axis is the result R as shown in Figure 2.4a. In the case of point doubling a tangent is drawn at P and then a reflection of the intersection point is the result R as shown in Figure 6.2.



(a) Addition: $P + Q = R$. **(b)** Doubling: $2P = R$

Figure 2.4: Addition and doubling of elliptic curve points

The Weierstrass equation can be simplified for non-supersingular curve $E/\mathbb{F}_{2^m} : y^2 + xy = x^3 + ax^2 + b$. The following are the group laws:

- *Identity.* $P + \infty = \infty + P = P$ for all $P \in E(\mathbb{F}_2^m)$.
- *Negatives.* If $P = (x, y) \in E(\mathbb{F}_2^m)$, then $-P = (x, x + y) \in \mathbb{F}_{2^m}$ and $P - P = \infty$.
- *Point addition.* Let $P = (x_1, y_1) \in E(\mathbb{F}_{2^m})$ and $Q = (x_2, y_2) \in E(\mathbb{F}_{2^m})$, Then $P + Q = (x_3, y_3)$, where $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ and $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$ with $\lambda = (y_1 + y_2)/(x_1 + x_2)$.

- *Point doubling.* Let $P = (x_1, y_1) \in E(\mathbb{F}_{2^m})$, where $P \neq -P$. Then $2P = (x_3, y_3)$, where $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ and $x_3 = \lambda^2 + \lambda + a = {x_1}^2 + \frac{b}{{x_1}^2}$ and $y_3 = {x_1}^2 + \lambda x_3 + x_3$ where $\lambda = x_1 + y_1/x_1$.

2.3.4 Point representation

The coordinates (x, y) considered in 2.3.3 are *affine* coordinates. In the same section point addition and point doubling are shown using affine coordinates. Field inversion and multiple field multiplication are needed for point addition or doubling when affine coordinates are used. Inversion free algorithms can be implemented with the use of *Projective Coordinates* [8]. Projective coordinates consists of more than two terms in their coordinates e.g., (x, y, z). Equivalence relation of two projective coordinates is defined as follows:

$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2)$ if $X_1 = \lambda^c X_2, Z_1 = \lambda^d Y_2, Z_1 = \lambda Z_2$ for some $\lambda \in K^*$. The projective points where $Z = 0$ is called the *line at infinity*. The one-to-one correspondence between affine and projective coordinates can be achieved by having $Z = 1$ and in fact it is the point $(X/Z^c, Y/Z^d, 1)$. If $c = 1$ and $d = 1$ then this is called *standard projective coordinates* and the point at infinity is $(0 : 1 : 0)$.

2.3.5 Point operations

The curve $y^2 = x^3 + ax + b$ becomes $Y^2Z = X^3 + aX^2Z + bZ^3$ when using projective coordinates. Let $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$ are the points to add, and $R = (X_R, Y_R, Z_R)$ is the resultant point. Point addition and point doubling algorithms of projective coordinates in this curve are presented in [12]. Algorithm 2.6 shows that point addition algorithm in standard projective coordinates.

Algorithm 2.7 shows how can point doubling algorithm be done. Note that in both - point doubling and point addition - algorithms use addition which is essentially an XOR ($\oplus$) in addition to multiplication and squaring where the binary field multiplication and squaring algorithms explained earlier are used.

Algorithm 2.6 Point Addition in standard projective coordinates.

procedure PointAdd(P, Q)
  if $P = \infty$ **then**
    return Q.
  end if
  if $Q = \infty$ **then**
    return P.
  end if
  $S_1 \leftarrow Y_P \cdot Z_Q$.
  $S_2 \leftarrow X_P \cdot Z_Q$.
  $A \leftarrow S_1 + Z_P \cdot Y_Q$.
  $B \leftarrow S_2 + Z_P \cdot X_Q$.
  $S_3 \leftarrow A + B$.
  $C \leftarrow B^2$.
  $D \leftarrow Z_P \cdot Z_Q$.
  $E \leftarrow B \cdot C$.
  $F \leftarrow (A \cdot S_3 + b \cdot C) \cdot D + E$.
  $X_R \leftarrow B \cdot F$.
  $Y_R \leftarrow C \cdot (A \cdot S_2 + BS_1) + S_3F$.
  $Z_R \leftarrow E \cdot D$.
  return R.
end procedure

Algorithm 2.7 Point doubling in standard projective coordinate.

1: **procedure** POINTDOUBLE(P)
2: **if** $Q = \infty$ **then**
3: **return** Q.
4: **end if**
5: $A \leftarrow {X_P}^2$.
6: $B \leftarrow A + Y_P \cdot Z_P$.
7: $C \leftarrow X_P \cdot Z_P$.
8: $BC \leftarrow B + C$.
9: $D \leftarrow C^2$.
10: $E \leftarrow B \cdot BC + b \cdot D$.
11: $X_R \leftarrow C \cdot E$.
12: $Y_R \leftarrow BC \cdot E + A^2 \cdot C$.
13: $Z_R \leftarrow C \cdot D$.
14: **return** R.
15: **end procedure**

2.3.6 Point multiplication

Consider the problem $Q = kP$ where P is a point and k is an integer. The result Q is another point on the elliptic curve K over the field $\mathbb{F}_{2^m}$, this is called *point multiplication* or *scalar multiplication*. The problem to find k given P and Q is called *elliptic curve discrete logarithm problem* which is believed to be intractable, in general. Algorithm 2.8 shows the scalar multiplication algorithm. It is basically double and add algorithm. The algorithms described above for point doubling and point addition can be used.

NAF representation and window method

If we can rewrite k such that $k_i \in \{0, \pm 1\}$ and no two consecurtvie digits k_i are not zero, then the result is called non-adjacent form (NAF). Therefore, when building the point multiplication algorithm and using NAF representation, we subtract the value of P from the result in case the value of k_i is -1. The idea could be extended to a *width-w* NAF. In this case, k_i is an odd integer such that $|k_i| < 2^{w-1}$ and no two consecutive nonzero digits. In this case, we add or subtract multiple of the point according to the value of k_i digit.

Algorithm 2.8 Right-to-left binary method for point multiplication.

```
1: procedure PointMultiply(k, P)                ▷ k = (k_{t-1}, ..., k_1, k_0)_2, P ∈ E(F_{2^m}).
2:     Q ← ∞.
3:     for i from 0 to t − 1 do
4:         if k_i = 1 then
5:             Q ← Q + P.
6:         end if
7:         P ← 2P.
8:     end for
9:     return Q.
10: end procedure
```

Sliding window method uses the same NAF representation but it traces multiple digits to form an odd positive or negative number and then add or subtract multiple of the point. To subtract a point you add its negative so if we have $P = (x, y)$. Then, $-P = (x, x + y)$. Algorithm 2.9 shows how we can get the NAF representation of an integer. Algorithm 2.10 shows the sliding window point multiplication algorithm.

2.4 Pairing

In this section, pairing used in cryptography is briefly reviewed. First, a definition of bilinear pairing is given. Then, Tate pairing is discussed. Finally, it is shown how bilinear pairings can be satisfied from the Tate pairing. Refer to [4] for more information.

2.4.1 Bilinear pairing

Let $G_1 = \langle P \rangle$ be an additively-written group of order n with ∞ as identity. Also let G_T be a multiplicatively-written group of order n and identity 1. A *bilinear pairing* on (G_1, G_T) is a map $\hat{e} : G_1 \times G_1 \to G_T$ that satisfy these conditions:

- (*bilinearity*) For all $R, S, T \in G_1, \hat{e}(R + S, T) = \hat{e}(R, T)\hat{e}(S, T)$ and $\hat{e}(R, S + T) = \hat{e}(R, S)\hat{e}(R, T)$.

Algorithm 2.9 Computing the NAF of a positive integer.

```
1: procedure GetNAF(k)                          ▷ k is a positive Integer.
2:     i ← 0.
3:     while k ≥ 1 do
4:         if k is odd then
5:             k_i ← 2 − (k mod (4)).
6:             k ← k − k_i.
7:         else
8:             k ← 0.
9:             i ← i + 1.
10:        end if
11:    end while
12:    return (k_{i−1}, k_{i−2}, ..., k_1, k_0).
13: end procedure
```

- (*non-degeneracy*) $\hat{e}(P, P) \neq 1$.
- (*computability*) $\hat{e}$ can be computed efficiently.

2.4.2 Tate pairing

Let $E[n]$ be the set of all points $P \in E(\overline{K})$ where $\overline{K}$ is the closure of K and $K = \mathbb{F}_q$. Let μ_n denote order-n subgroup of ${\mathbb{F}_q}^*$. Then, the Tate pairing is a map $e : E[n] \times E[n] \rightarrow \mu_n$ which can be defined in the following way: Let $P, Q \in E[n]$ and f_P is a function such that $div(f_P) = n(P) - n(\infty)$ where div is the divisor. Let $R \in E[n]$ such that $R \notin \{\infty, P, -Q, P - Q\}$ and $D_Q = (Q + R) - (R)$ and let $D_Q = (Q + R) - (R)$. Then, $e(P, Q) = f_P(D_Q)^{(q^k-1)/n} = (\frac{f_P(Q+R)}{f_P(R)})^{(q^k-1)/n}$.

2.5 Hyperelliptic Curves

Hyperelliptic curves (HEC) are a generalization of elliptic curves. If the genus of a Hyperelliptic curve $g = 1$ then the curve is an elliptic curve. For every genus $g \geq 1$, there is a hyperelliptic curve. In this section, a short introduction to hyperelliptic curve is provided.

Algorithm 2.10 Point multiplication using sliding window method.

1: **procedure** SLIDINGWINDOWPOINTMULTIPLY($P, kNaf$)
2: Compute $P_i = iP$ for $i \in \{1, 3, \ldots, 2(2^w - (-1)^w)/3 - 1\}$
3: $Q \leftarrow \infty, i \leftarrow l - 1$.
4: **while** $i \geq 0$ **do**
5: **if** $k_i = 0$ **then**
6: $t \leftarrow 1, u \leftarrow 0$.
7: **else**
8: find the largest $t \leq w$ such that $u \leftarrow (k_i, \ldots, k_{i-t+1})$ is odd.
9: **end if**
10: $Q \leftarrow 2^t Q$.
11: **if** $u > 0$ **then**
12: $Q \leftarrow Q + P_u$.
13: **else if** $u < 0$ **then**
14: $Q \leftarrow Q - P_{-u}$.
15: **end if**
16: $i \leftarrow i - t$.
17: **end while**
18: **return** (Q).
19: **end procedure**

Menezes, Wu and Zuccherato report gives an excellent introduction to hyperelliptic curves [5].

A *hyperelliptic curve* C of genus g over the field K satisfy the following equation

$$C : v^2 + h(u)v = f(u) \text{ in } K[u, v],$$

where $h(u) \in k[u]$ is a polynomial of degree at most g and $f(u) \in K[u]$ is a monic polynomial of degree $2g + 1$. Assuming that $\overline{K}$ is the algebraic closure of K, then there is no solution $(u, v) \in \overline{K} \times \overline{K}$ that satisfies the equation $v^2 + h(u)v = f(u)$ and its partial derivative equation (with respect to v and u): $2v + h(u) = 0$ and $h'(u)v - f'(u) = 0$. If a solution exists, then this is called a *singular point*.

Assume that L is an extension field of K. Then, the set of *L-rational points* on C, denoted $C(L)$, is the set of all points $P = (x, y) \in L \times L$ that satisfy the hyperelliptic curve equation in addition to the point at infinity ∞. The opposite of P is the point $\tilde{P} = (x, -y - h(x))$. The point is special if $P = \tilde{P}$.

An example of hyperelliptic curve of genus $g = 2$ and $h(u) = 0$ is $C_1 : v^2 = u^5 + u^4 + 4u^3 + 3u + 3$. Another example where $g = 2$, $h(u) = u$ and $f(u) = u^5 + 5u^4 + 6u^2 + u + 3$ is $C_2 : v^2 + uv = u^5 + 5u^4 + 6u^2 + u + 3$ over the finite field $\mathbb{Z}_7$ then the $\mathbb{Z}_7$-rational points are $C(\mathbb{Z}_7) = \{\infty, (1, 1), (1, 5), (2, 2), (2, 3), (5, 3), (5, 6), (6, 4)\}$.

Chapter 3

Improved Polynomial Multiplication and Point Addition/Doubling Algorithms

In this chapter, we present a survey of the major cryptographic algorithms related to our work. The first section surveys several polynomial multiplication algorithms. The second section highlights the improved point addition and point doubling algorithms in binary elliptic curves.

3.1 Polynomial Multiplication Algorithms

In this section, we first introduce the Karatsuba Algorithm, more specifically, the 2-way split (K2W) algorithm. After that a number of 3-way algorithms are investigated. Most of the material of this section is based on the work in [7] [13]. Next, we look at a general methodology to design a 3-way split algorithm. Then, the Karatsuba 3-way split (K3W) formulæ are presented. After that, we show an improvement on the Karatsuba 3-way (IK3W) formulæ. Then, Bernstein's 3-way split (B3W) formulæ are highlighted. Finally, the field extension based 3-way split formulæ (FE3W) are presented.

3.1.1 Karatsuba (K2W) multiplication

The idea of Karatsuba polynomial multiplication is to split the polynomial in two half-size polynomials and recursively call the algorithm in order to reduce the complexity of multiplication [14]. Assume that we have the following two polynomials, $A(X) = \sum_{i=0}^{m-1} A_i X^i$ and $B(X) = \sum_{i=0}^{m-1} B_i X^i$. The polynomials can be rewritten in the following way: $A(X) = A_1 X + A_0$ and $B(z) = B_1 X + B_0$ (after substituting $Y = X^{m/2}$ and then replacing Y with X). The following auxiliary variables are needed: D_0, D_1, and $D_{0,1}$, where $D_0 = A_0 \cdot B_0, D_1 = A_1 \cdot B_1$, and $D_{0,1} = (A_0 + A_1) \cdot (B_0 + B_1)$. Then we can compute the polynomial $C(X) = A(X)B(X)$ by the following reconstruction: $C(z) = D_1 X^2 + (D_{0,1} - D_0 - D_1)X + D_0$.

3.1.2 Sketching 3-way split multiplication algorithm

Formulas that consist of multi-evaluation and interpolation (Toom-Cook like [15] [16]) can be used to design 3-way split algorithms. Assume that $A(X)$ and $B(X)$ are polynomials of degree $n-1$ in $\mathcal{R}[X]$ where $\mathcal{R}$ is a ring and n is a power of 3. We can split these polynomials into three parts: $A = A_0 + A_1 X^{n/3} + A_2 X^{2n/3}$ and $B = B_0 + B_1 X^{n/3} + B_2 X^{2n/3}$. The degrees of A_i and B_i are both $n/3 - 1$. By substituting $X^{n/3}$ by Y, we can rewrite A and B as: $A = A_0 + A_1 Y + A_2 Y^2$ and $B = B_0 + B_1 Y + B_2 Y^2$. Since A and B are both polynomials of degree 2 in terms of Y, then their product C is of degree 4 and it has five terms. Therefore, polynomial C can be determined when we compute its value at five points. Let the points be $\alpha_1, \ldots, \alpha_4 \in \mathcal{R}$ and $\alpha_5 = \infty$; then the multi-evaluation can be done term by term for $A(\alpha_i)$ and $B(\alpha_i)$ and the product can be computed as $C(\alpha_i)$ at the five points. Then, interpolation can be used to compute $C(Y)$ by employing the Lagrange polynomial $L_i(Y) = \prod_{j=1, j\neq i}^{4} \frac{Y - \alpha_i}{\alpha_i - \alpha_j}$ for $i = 1, \ldots, 4$ and $L_\infty = \prod_{i=1}^{4} (Y - \alpha_i)$. Finally, to compute $C(Y)$, the following formula can be used: $C(Y) = \sum_{i=1}^{4} C(\alpha_i) L_i(Y) + C(\infty) L_\infty(Y)$. To get the result in terms of X, Y is substituted by $X^{n/3}$.

3.1.3 The Karatsuba 3-way split (K3W) formulæ

Assume the same $A(X)$ and $B(X)$ of degree $n-1$ considered in the previous section but the field in this case is $\mathbb{F}_2[X]$. However, in this field, there are only two points, 0 and 1, to

be used in multi-evaluation. The Winograd method [17] suggested the replacement of the two missing points by multiplication modulo Y^2+Y+1. So the multi-evaluation is as the following:

$$\begin{cases} C(0) &= A(0)B(0) = A_0B_0, \\ C(1) &= A(1)B(1) = (A_0+A_1+A_2)(B_0+B_1+B_2), \\ C(Y) &= (A_0+A_2+(A_1+A_2)Y)(B_0+B_2+(B_1+B_2)Y) \mod (Y^2+Y+1), \\ C(\infty) &= A(\infty)B(\infty) = A_2B_2. \end{cases}$$

To perform the multiplication modulo Y^2+Y+1, Winograd used the Karatsuba formula which needs 3 multiplications $(A_0+A_1)(B_0+B_1)$, $(A_1+A_2)(B_1+B_2)$, and $(A_0+A_2)(B_0+B_2)$. The Chinese remainder theorem can be used for the reconstruction. The following are the formulæ used for the recursive products.

$$\begin{cases} P_0 &= A_0B_0 \\ P_1 &= A_1B_1 \\ P_2 &= A_2B_2 \\ P_3 &= (A_0+A_1)(B_0+B_1) \\ P_4 &= (A_1+A_2)(B_1+B_2) \\ P_5 &= (A_0+A_2)(B_0+B_2) \end{cases}$$

The following formulæ can be used for reconstruction.

$$\begin{cases} R_0 &= P_0+P_1 \\ R_1 &= P_3+R_0 \\ R_2 &= P_0+P_1+P_2+P_5 \\ R_3 &= P_1+P_2+P_4 \\ C &= P_0+R_0X^{n/3}+R_1X^{2n/3}+R_2X^{3n/3}+P_2X^{4n/3} \end{cases}$$

3.1.4 Improved Karatsuba 3-way (IK3W) formulæ

The reconstruction process can be re-arranged to save some computations to the 3-way algorithm presented in the previous section. Assume we have the same A and B polynomials used in the previous section which are split in three parts. The product formulæ can

be used in a similar fashion to the 3-way formulæ. The following expression is used in the previous section for the reconstruction:

$$C = P_0 + (P_0 + P_1 + P_3)X^{n/3} + (P_0 + P_1 + P_2 + P_5)X^{2n/3} + (P_1 + P_2 + P_4)X^{3n/3} + P_2X^{4n/3}$$

The above expression can be rewritten in the following way:

$$C = (P_0 + X^{n/3}P_1 + X^{2n/3}P_2)(1 + X^{n/3} + X^{2n/3}) + P_3X^{n/3} + P_5X^{2n/3} + P_4X^{3n/3}$$

We can define $R_0 = P_0 + X^{n/3}P_1 + X^{2n/3}P_2$ and $R_1 = R_0(1 + X^{n/3} + X^{2n/3})$. Then we have C in terms of P_i and R_i as $C = R_1 + P_3X^{n/3} + P_5X^{2n/3} + P_4X^{3n/3}$. Therefore the product formulæ are shown below:

$$\begin{cases} P_0 &= A_0B_0 \\ P_1 &= A_1B_1 \\ P_2 &= A_2B_2 \\ P_3 &= (A_0 + A_1)(B_0 + B_1) \\ P_4 &= (A_1 + A_2)(B_1 + B_2) \\ P_5 &= (A_0 + A_2)(B_0 + B_2) \end{cases}$$

The reconstruction formulæ are as follows:

$$\begin{cases} R_0 &= P_0 + X^{n/3}P_1 + X^{2n/3} \\ R_1 &= R_0(1 + X^{n/3} + X^{2n/3}) \\ C &= R_1 + P_3X^{n/3} + P_5X^{2n/3} + P_4X^{3n/3} \end{cases}$$

3.1.5 Bernstein's (B3W) formulæ

Multi-evaluation and interpolation are also used in the Bernstein algorithm [18] by evaluating the polynomials at $0, 1, X, X + 1$, and ∞. So the pairwise product of the evaluations of $A(Y)$ and $B(Y)$ are as follows:

$$\begin{cases} P_0 &= A_0B_0 \\ P_1 &= (A_0 + A_1 + A_2)(B_0 + B_1 + B_2) \\ P_2 &= (A_0 + A_1X + A_2X^2)(B_0 + B_1X + B_2X^2) \\ P_3 &= ((A_0 + A_1 + A_2) + (A_1X + A_2X^2)) \times ((B_0 + B_1 + B_2) + (B_1X + B_2X^2)) \\ P_4 &= A_2B_2 \end{cases}$$

The following expressions are proposed by Bernstein for reconstruction:

$$\begin{cases} U &= P_0 + (P_0 + P_1)X \\ V &= P_2 + (P_2 + P_3)(X^{n/3} + X) \\ C &= U + P_4(X^{4n/3} + X^{n/3}) + \frac{(U+V+P_4(X^4+X))(X^{2n/3}+X^{n/3})}{X^2+X} \end{cases}$$

The explicit computation of Bernstein is composed of three parts: multi-evaluation, product and reconstruction. The multi-evaluation formulæ are:

$$\begin{cases} M_1 &= A_0 + A_1 + A_2, & M_1^{'} &= B_0 + B_1 + B_2 \\ M_2 &= A_1X + A_2X^2, & M_2^{'} &= B_1X + B_2X^2 \\ M_3 &= A_0 + M_2, & M_3^{'} &= B_0 + M_2^{'} \\ M_4 &= M_1 + M_2, & M_4^{'} &= M_1^{'} + M_2^{'} \end{cases}$$

The product formulæ are shown below:

$$\begin{cases} P_0 &= A_0B_0 \\ P_1 &= M_1M_1^{'} \\ P_2 &= M_3M_3^{'} \\ P_3 &= M_4M_4^{'} \\ P_4 &= A_2B_2 \end{cases}$$

The reconstruction formulæ are as follows:

$$\begin{cases} S &= P_2 + P_3 \\ U &= P_0 + (P_0 + P_1)X^{n/3} \\ V &= P_2 + S(X^{n/3} + X) \\ W &= U + V + P_4(X^4 + X) \\ W^{'} &= W/(X^2 + X) \\ W^{''} &= W'(X^{2n/3} + X^{n/3}) \\ C &= U + P_4(X^{4n/3} + X^{n/3}) + W^{''} \end{cases}$$

3.1.6 Field extension based 3-way (FE3W) formulæ

In the previous section, we have seen that Bernstein makes the evaluation at X and $X+1$ as the two extra points other than 0 and 1 values available in $\mathbb{F}_2$ in addition to ∞ point.

In [7], as an alternative method, field extension $\mathbb{F}_4 = \mathbb{F}_2[\alpha]/(\alpha^2+\alpha+1)$ has been used. In this case, the polynomial can be evaluated at $0, 1, \alpha, \alpha+1$, and ∞. As a result, we have the following recursive multiplication:

$$\begin{cases} P_0 &= A_0B_0 & \text{in } \mathbb{F}_2[\mathrm{X}] \\ P_1 &= (A_0+A_1+A_2)(B_0+B_1+B_2) & \text{in } \mathbb{F}_2[\mathrm{X}] \\ P_2 &= (A_0+A_2+\alpha(A_1+A_2))(B_0+B_2+\alpha(B_1+B_2)) & \text{in } \mathbb{F}_4[\mathrm{X}] \\ P_3 &= (A_0+A_1+\alpha(A_1+A_2))(B_0+B_1+\alpha(B_1+B_2)) & \text{in } \mathbb{F}_4[\mathrm{X}] \\ P_4 &= A_2B_2 & in\ \mathbb{F}_2[X] \end{cases}$$

The Lagrange interpolation can be used to compute the reconstruction $C = A \times B$ as shown below:

$$C = (P_0 + X^{n/3}P_4)(1+X^n) + (P_1 + (1+\alpha)(P_2+P_3))(X^{n/3}+X^{2n/3}+X^n)$$

The following are the formulæ for the multi-evaluation, which is similar in both cases in $\mathbb{F}_2$ and in $\mathbb{F}_4$.

$$\begin{cases} M_1 = A_0+A_1, & M_1' = B_0+B_1 \\ M_2 = A_1+A_2, & M_2' = B_1+B_2 \\ M_3 = \alpha M_2, & M_3' = \alpha M_2' \\ M_4 = M_1, & M_4' = M_1'+M_3' \\ M_5 = M_4+M_2, & M_5' = M_4'+R_2' \\ M_6 = M_1+A_2, & M_6' = M_1'+B_2 \end{cases}$$

The following are the product formulæ

$$\begin{cases} P_0 = A_0B_0 \\ P_1 = M_6M_6' \\ P_2 = M_5M_5' \\ P_3 = M_4M_4' \\ P_4 = A_2B_2 \end{cases}$$

The following are the reconstruction formulæ in $\mathbb{F}_4$:

$$\begin{cases} U_1 &= P_2 + P_3 \\ U_2 &= \alpha U_1 \\ U_3 &= (1+\alpha)U_1 \\ U_4 &= P_1 + U_3 \\ U_5 &= U_4(X^{n/3} + X^{2n/3} + X^{3n/3}) \\ U_6 &= P_0 + X^{n/3}P_4 \\ U_7 &= U_6(1+X^n) \\ C &= U_7 + U_5 + X^n U_2 + P_2 X^{2n/3} + P_3 X^{n/3} \end{cases}$$

If $a = a_0 + a_1\alpha$ and $b = b_0 + b_1\alpha$, then we can denote $[a+b]_{const} = a_0 + b_0$. The following formulæ show the reconstruction in $\mathbb{F}_2$:

$$\begin{cases} U_1 &= P_2 + P_3 \\ U_2 &= [\alpha U_1]_{const} \\ U_3 &= [(1+\alpha)U_1]_{const} \\ U_4 &= [P_1 + U_3]_{const} \\ U_5 &= [U_4(X^{n/3} + X^{2n/3} + X^{3n/3})]_{const} \\ U_6 &= [P_0 + X^{n/3}P_4]_{const} \\ U_7 &= [U_6(1+X^n)]_{const} \\ C &= [U_7 + U_5 + X^n U_2 + P_2 X^{2n/3} + P_3 X^{n/3}]_{const} \end{cases}$$

3.1.7 Reducing two multiplications to one $\mathbb{F}_4[X]$ multiplication

The field extension 3-way split algorithm presented in the previous section can be used to replace two $\mathbb{F}_2$ multiplications AB and AC such that $A, B, C \in \mathbb{F}_2[X]$ by one $\mathbb{F}_4$ multiplication. We can write $P = A(B + \alpha C) = P_0 + \alpha P_1$, this results in $P_0 = AB$ and $P_1 = AC$. This shows some potential advantage since it is faster to use a single multiplication in $\mathbb{F}_4[X]$ than two multiplications in $\mathbb{F}_2[X]$. The use of a single multiplication in $\mathbb{F}_4[X]$ works here as simultaneous finite field operations which include the products AB and AC in $\mathbb{F}_2[X]$. This kind of computation exists in elliptic curve point addition and point doubling algorithms.

3.2 ECC Algorithms

As mentioned in Chapter 2 the Weierstrass form of elliptic curve over $\mathbb{F}_2$ is $y^2 + xy = x^3 + a_2x^2 + a_6$ where $a_2, a_6 \in \mathbb{F}_{2^n}$ and $a_6 \neq 0$. A point (x, y) on an elliptic curve can be expressed as (X, Y, Z) where $x = X/Z$ and $y = Y/Z$. The projective representation of the curve is $Y^2Z + XYZ = X^3 + a_2X^2Z + a_6Z^3$. This equation of the curve is used in the following discussion.

3.2.1 Point addition

If we want to add two points $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, Z_2)$, then their addition is $R = P + Q = (X_3, Y_3, Z_3)$. The following are the formulæ for point addition [12]. We will call this point addition *conventional* point addition algorithm in the rest of the proposal.

$$\begin{cases} S_1 = Y_1Z_2, \quad S_2 = X_1Z_2, \quad A = S_1 + Z_1Y_2, \\ B = S_2 + Z_1X_2, \quad S_3 = A + B, \quad C = B^2 \\ D = Z_1Z_2, \quad E = BC, \quad F = (AS_3 + a_2C)D + E \\ X_3 = BF, \quad Y_3 = C(AS_2 + BS_1) + S_3F, \quad Z_3 = ED \end{cases}$$

In the previous set of formulæ, we have Z_2 as common for the products $S_1 = Y_1Z_2$ and $S_2 = X_1Z_2$. Moreover, Z_1 is common for the products Z_1Y_2 and Z_1X_2. A common operand A is for AS_3 and AS_2. $(AS_3 + a_2C)D$ and ED have D in common. The common term F is there for FB and FS_3. The following formula can be used as new point addition formulæ [6]. We will call these forumulas CANH point addition (after the last names of the authors of [6]: Cenk, M., Alrefai, A. S., Negre, C. and Hasan, M. A.)

$$\begin{cases} T_1 = Z_2(Y_1 + \alpha X_1), \quad T_2 = T_{1,0}, \quad T_3 = T_{1,1}, \\ T_4 = Z_1(Y_2 + \alpha X_2), \quad T_5 = T_{4,0}, \quad T_6 = T_{4,1}, \\ T_7 = T_2 + T_5, \quad T_8 = T_3 + T_6, \quad T_9 = T_7 + T_8, \quad T_{10} = T_8^2, \\ T_{11} = Z_1Z_2, \quad T_{12} = T_8T_{10}, \quad T_{13} = T_7(T_9 + \alpha T_3), \\ T_{14} = T_{13,0}, \quad T_{15} = T_{13,1}, \quad T_{16} = T_{14} + a_2T_{10}, \\ T_{17} = T_{11}(T_{16} + \alpha T_{12}), \quad T_{18} = T_{17,0}, \quad T_{19} = Z_3 = T_{17,1}, \\ T_{20} = T_{18} + T_{12}, \quad T_{21} = T_{20}(T_8 + \alpha T_9), \quad T_{22} = T_{21,0}, \\ T_{23} = T_{21,1}, \quad X_3 = T_{22}, \quad Y_3 = T_{10}(T_{15} + T_8T_2) + T_{23}. \end{cases}$$

3.2.2 Point doubling

The best known formulæ for point doubling as presented in [12] are shown below. We call these formulæ conventional point doulbling:.

$$\begin{cases} A = X_1^2,\ B = A + Y_1 Z_1,\ C = X_1 Z_1, D = B + C,\ E = C^2, \\ F = BD + a_2 E,\ X_3 = CF,\ Y_3 = DF + A^2 C,\ Z_3 = CE. \end{cases}$$

The products $Y_1 Z_1$ and $X_1 Z_1$ have Z_1 in common. CF and DF have F in common. Moreover, $A^2 C$ and CE have C as a common operand. The following are the formulæ after applying the modifications [6]. These formulæ are called in this proposal CANH point doubling after the names of the authors in [6].

$$\begin{cases} T_1 = X_1^2,\ T_2 = Z_1(Y_1 + \alpha X_1),\ T_3 = T_{2,0},\ T_4 = T_{2,1}, T_5 = T_1 + T_3,\ T_6 = T_4 + T_5, \\ T_7 = T_4^2,\ T_8 = T_5 T_6 + a_2 T_7, T_9 = T_8(T_4 + \alpha T_6),\ T_{10} = T_{9,0},\ T_{11} = T_{9,1}, \\ T_{12} = T_1^2,\ T_{13} = T_4(T_{12} + \alpha T_7),\ T_{14} = T_{13,0},\ T_{15} = T_{13,1}, \\ X_3 = T_{10},\ Y_3 = T_{11} + T_{14},\ Z_3 = T_{15}. \end{cases}$$

3.3 Related Work

In [19], different finite fields for elliptic curve cryptosystems are compared. They found that the use of optimized extension fields (OEFs) produces greater performance. An efficient algorithm for multiplication in $\mathbb{F}_{2^m}$ is described in [20]. The authors proposed a comb method with the use of a window as an improvement over the shift and add method.

An excellent survey of the techniques used to multiply elements in different rings is presented in [21]. This covers Karatsuba and Toom multiplications and different tricks for performing multiplication. Their paper focuses both on multiplying large integers and finding the product of polynomial over a commutative ring. In [22], a proposed method to improve splitting of input is described. This method improves the theoretical XOR gate delay of the Karatsuba multiplier. In [23], the toeplitz matrix-vector products and coordinate transformation techniques are utilized to achieve a subquadratic space complexity parallel multiplier.

Bernstein proposed an optimization of Karatsuba forumula for binary polynomial multiplication in [18] by rearranging the reconstruction part in two recursions of the formula. A generalization of this approach is propsed by Negre in [24], who extended his work in [25] to three-way split formula. His work improves over the best known space complexity of [7] while having the same time complexity as the best known approach in [22].

The work in [26] studies the optimization effects of optimizing software implementation on small binary field arithmetic. Their implementation smooths the performance of binary fields to better resemble theoretical results. They noted that their results might require the development of new explicit formulæ for arithmetic on elliptic and hyperelliptic curves. The use of the vector instruction set in the software implementation of binary field arithmetic is described in [27]. Their representation uses extensively the parallel lookup instruction introduced in desktop platforms which expedite the implementation of cryptographic algorithms. For large finite fields GF(2^n), the work of Luo J. et al. studies efficient software implementation techniques as well as present new optimization methodology [28]. The reader might refer to [29] for superb article that survey different techniques for implementing finite field arithmetic in software. The work of A. Reyhani-Masoleh in [30] present efficient algorithms for field multiplication in normal basis.

The work of Lopez and Dahab in [31] proposes an optimized version of the work in [32] for doing fast scalar multiplication without precomputation. Higuchi and Takagi proposed in [33] a fast addition algorithm using projective coordinates. Their algorithm improves over the Lopez and Dahab algorithm proposed in [34] by the use of the same coordinate system.

A study of software implementation of NIST recommended elliptic curves is presented in [35]. Their results show that projective coordinates are better than affine coordinates due to the use of costly inversion. The use of Koblitz curves is better than random curves in their implementation. The work of Bernstein and Lange [36] studies how to optimize single-scalar multiplication in elliptic curves. This includes how many points are needed for precomputation in sliding window computation of scalar multiplication and other issues.

The use of instruction set extension to speed up arithmetic over binary fields is discussed in [37] and [38]. For an excellent survey for fast hardware implementation of elliptic curve cryptography, the interested reader might refer to [39]. In [40] the authors described the design of a crypto-processor for ECC. Their design present high performance implementation that utilizes low area. The work in [41] presents the use of the parallelism in

residue number system to build a fast point multiplier in elliptic curve cryptography.

Wollinger et al. [42] focused on embedded processors in their study of elliptic curve cryptography and hyperelliptic curve cryptography and how it is affected by different architectures, processor types and resources. They improved the HECC algorithms.

3.4 Summary

The use of fast polynomial multiplication algorithms helps improve elliptic curve operations that depend on them. In this chapter, polynomial multiplication formulæ as well as elliptic curves point addition and point doubling formulæ have been investigated. Recursive multiplication algorithms depend on splitting the polynomials and dealing with the resulting smaller size polynomials. The FE3W algorithm can replace two multiplications in the same field that have a common factor. This idea has been used to improve point addition and doubling algorithms in elliptic curves, and the formulæ for the improved algorithms have been shown.

Chapter 4

Software Implementation

In this chapter, we introduce some algorithms used to develop the elliptic curve point multiplication to generate the results. More specifically, we discuss the implementation of 2-way and 3-way split algorithms for polynomial multiplication. These include the K2W [14], K3W [14], IK3W [7], B3W [18] and FE3W [7]. Also we discuss the implementation of the algorithms related to elliptic curve like point doubling, point addition and point multiplication [6].

4.1 Preliminaries

In this section we provide an introduction to software implementation and the development environment used. Then, we will talk about *Big Integer* structure that is used in our implementation. After that, the overall view of the program is presented. The description of polynomial multiplication algorithms are presented next. This is followed by the structure of the reduction algorithms and then the structure of elliptic curve algorithms. At the end, other classes implemented are introduced.

4.1.1 Development environment

$C\sharp$ is used as the programming language which is an improvement of $C++$ programming language but includes object oriented features like classes and methods. $C\sharp$ is part of .Net

framework. Visual studio is used as a development environment.

4.1.2 BigInteger

BigInteger structure is part of .Net 4.0 frameworks [43]. This structure accepts integers of any size. One can use it to perform all the basic operations used in integer. That includes normal operations like addition, subtraction, multiplication, and division and bit-wise operations like ORing, ANDing, and eXclusive ORing (XOR). The internal structure of BigInteger is an array of 32-bit integers. With the availability of this structure, one needs not to worry about the internal management of bits in this array and how the basic operation is handled. This makes the implementation of the algorithms a little bit easier.

4.1.3 High level view

Figure 4.1 shows the overall structure of the implementation. The main program is a class that is used to test the methods and to measure the timing results. Polynomial multiplication package or the set of classes contains the methods used to perform different types of polynomial multiplication algorithm. Reduction package contains algorithms to perform a general reduction algorithm or fast reduction algorithms for specific NIST fields. Elliptic Curve package contains classes to define elliptic curve points and the methods of point addition, point doubling and point multiplication.

4.1.4 Structure of polynomial multiplication

We implemented *five* polynomial multiplication algorithms. These are shown in Figure 4.2. They are mainly the K2W, K3W, IK3W, B3W and FE3W. Each algorithm is implemented in a class that extends polynomial multiplication interface. Later in this chapter, we will describe these algorithms in details.

4.1.5 Structure of reduction

Since our test results focus on NIST recommended curves, we have implemented faster reduction algorithms for these curves. Figure 4.3 shows the structure of the reduction

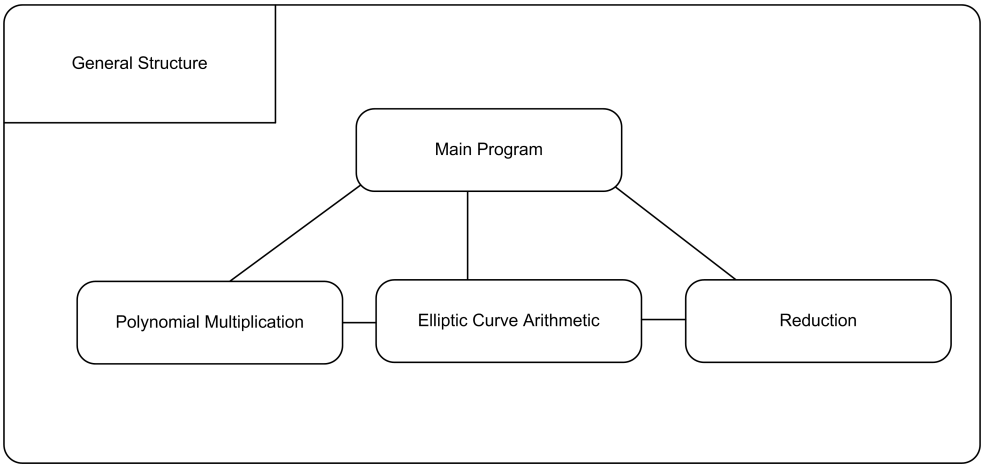


Figure 4.1: General structure of the implementation.

package. A method for general reduction and other reduction methods that work on elliptic curves: B163, B233, B283, B409 and B571.

4.1.6 Structure of elliptic curve arithmetic

To construct an elliptic curve, a class is built to represent the points. Each point has three coordinates X, Y, and Z. Figure 4.4 shows the elliptic curve package. ECPoint class has a method to determine whether the point is infinity (∞). The Elliptic curve class has the point addition and point doubling methods in addition to point multiplication methods.

4.1.7 Other classes

Some other classes are also built to complete the program. For example, one class is built to perform the inversion of an element in a field. This is used to convert the point from projective to affine coordinates in order to test the correctness of our implementation. Also configuration class is built to generate the table of results of polynomial multiplication of small numbers.

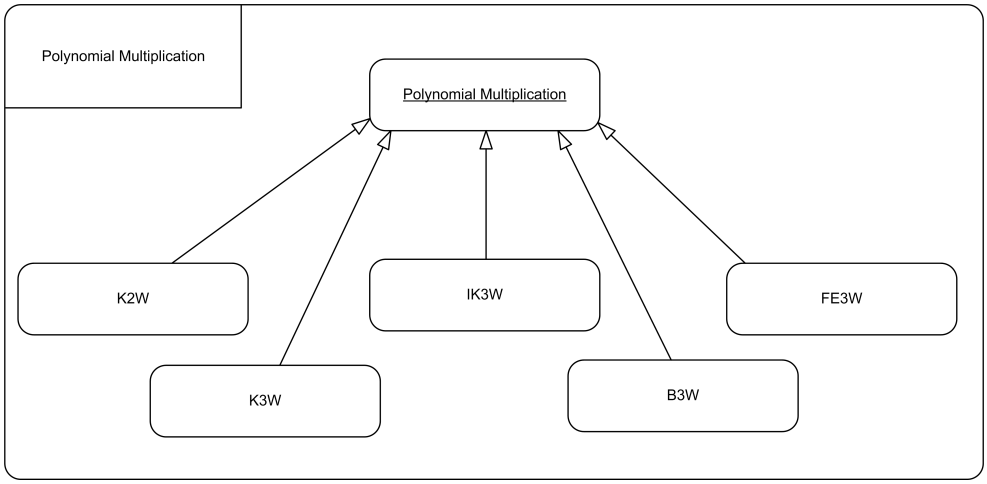


Figure 4.2: Structure of polynomial multiplication package.

4.2 Polynomial Multiplication

In this section, detail of implementation of the polynomial multiplication algorithms is presented. These algorithms include the K2W, K3W, IK3W , B3W and FE3W algorithms.

4.2.1 Polynomial multiplication algorithms

Algorithms of polynomial multiplication considered here are recursive in nature, i.e. a method inside an algorithm may call the method itself. These algorithms are based on the idea of splitting, similar to the Karatsuba algorithm reviewed in section 3.1.1. In that section the polynomial is divided into two subpolynomials. However, 3-way split methods divide the polynomial into three smaller polynomials.

In general there are five parts of polynomial multiplication algorithm as shown in Figure 4.5. The first one is *stop-check* in which if the polynomial size is less than a certain threshold then the check returns the result from a precomputed look up table. The second one is *splitting* in which the polynomial is split in two or three smaller polynomials. The third step is *multi-evaluation* which makes some operations on the polynomials and generate temporary variables to be used in the forth step which is *product recursive calls*. Multi-evaluation is actually a step to evaluate a number of points on the curve of product of the

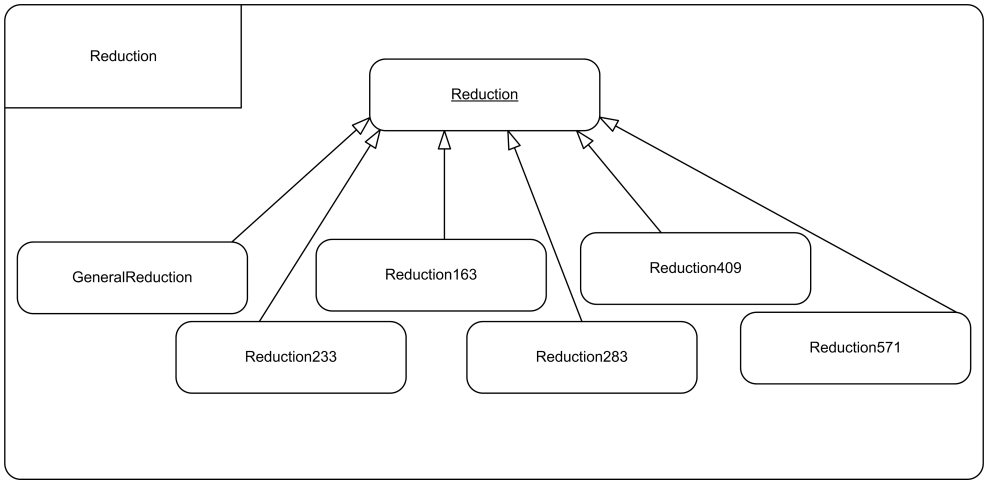


Figure 4.3: Structure of reduction package.

two polynomials. Product recursive calls usually call the same method again, but it uses as an input the sub-polynomials or the results of multi-evaluation step. The last step, which is called *reconstruction*, is the calculation conducted on the results of the recursive calls to get the resulting polynomial. Reconstruction is originally an interpolation step of the product results of the points in the curve. Multi-evaluation and reconstruction are known as Toom-Cook like formulas [7].

4.2.2 Look-up table

Polynomial multiplication over $\mathbb{F}_2$ is not a normal multiplication of two integers. It is actually a *carry-less* multiplication. The polynomials we are trying to multiply are of large degree, for example NIST fields go up to 570. Hence the degree of the product is up to 1140. Normal integer structure does not fit to represent polynomials. As a result, we use BigInteger structure, we described earlier in section 4.1.2, to represent them. Assume that we have two small polynomials we are trying to multiple $A = x^2 + x + 1$ and $B = x^2 + x$. The binary representations of these two polynomials are $(111)_2$ and $(110)_2$ respectively. These numbers are 7 and 6 in decimal and their radix-ten multiplication is 42. However, their polynomial multiplication is $(x^2 + x + 1) \cdot (x^2 + x) = x^4 + x^3 + x^2 + x^3 + x^2 + x = x^4 + x = (10010)_2 = (18)_{10}$. This example of carry-less multiplication is shown in Figure

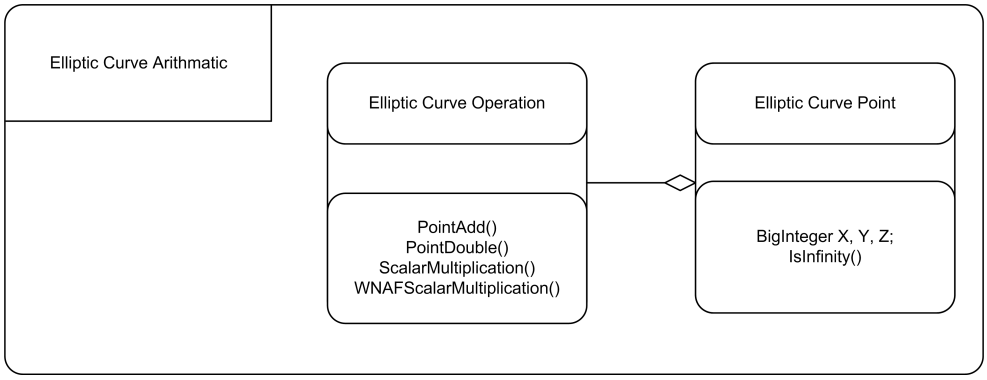


Figure 4.4: Elliptic curve package.

4.6 and it is different than conventional multiplication since the addition is done mod 2 or XOR ($\oplus$).

One can construct a look-up table (LUT) using a two dimensional array, where the rows of the array represent the first input and the columns represent the second input. The content of the cell in the array is the polynomial multiplication result. Table 4.1 is a polynomial multiplication look up table of input size up to 3 bits. One can get the result from the table by inserting the two input as index. For example, to multiply polynomials, whose decimal representation is 7 and 6, we index the table at row ($R = 7$) and column ($C = 6$) and the result ($R = 18$), we can write it as $LUT(7, 6) = 18$.

Table 4.1: Look up table for multiplication up to 3-bit input size.

R\C	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	8	10	12	14
3	0	3	6	5	12	15	10	9
4	0	4	8	12	16	20	24	28
5	0	5	10	15	20	17	30	27
6	0	6	12	10	24	30	20	18
7	0	7	14	9	28	27	18	21

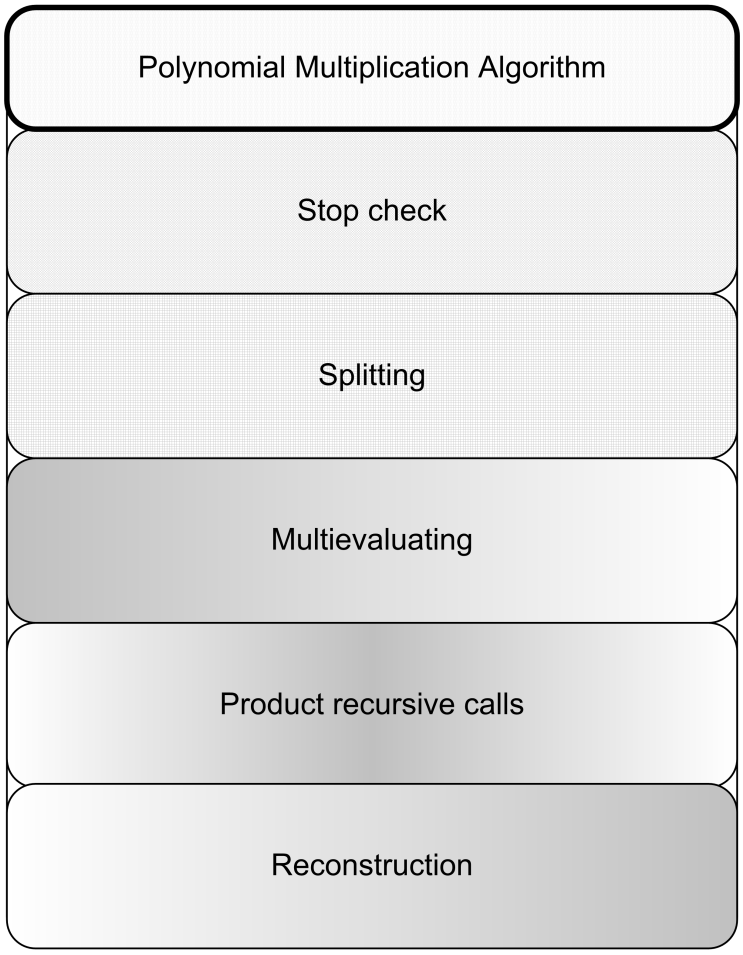


Figure 4.5: Polynomial multiplication algorithm structure.

4.2.3 Splitting

Since splitting is done in every algorithm, it is worth discussing it in a separate section. Algorithm 4.1 shows how a polynomial can be split in two subpolynomials using shift and XOR operations. Similarly, Algorithm 4.2 shows how to split the polynomial in three subpolynomials.

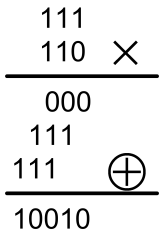


Figure 4.6: Carry-less multiplication.

Algorithm 4.1 Splitting into two subpolynomials.

procedure SPLITTWO(A, n) ▷ A is the polynomial to be split, n is subpolynomial length
  $A_1 \leftarrow A \gg n$.
  $A_0 \leftarrow A \oplus (A_1 \ll n)$.
  return (A_1, A_0).
end procedure

4.2.4 The K2W algorithm

In [44], Karatsuba suggested a method of performing multiplication of large integers. We assume the number of coefficients to be a power of 2. So the two polynomials that we intend to multiply are split into two halves. These halves can be used as if they were coefficients. Algorithm 4.3 shows the Karatsuba multiplication. Since the field is binary, XOR ($\oplus$) is used instead of addition and subtraction. Algorithm 4.3 shows the way it is implemented. Note that the use of *SplitTwo* is done in the same way as in Algorithm 4.1.

Algorithm 4.2 Splitting into three subpolynomials.

procedure THREEWAYSPLIT(A, n) ▷ A is two input polynomials, n is subpolynomial length
  $A_2 \leftarrow A \gg (n \ll 1)$. ▷ $n \ll 1$ is equal to $2 \cdot n$
  $A_1 \leftarrow (A \gg n) \oplus (A_2 \ll n)$.
  $A_0 \leftarrow A \oplus (A_2 \ll (n \ll 1)) \oplus (A_1 \ll n)$
  return (A_2, A_1, A_0).
end procedure

One can easily distinguish the degree of polynomial multiplication. The way we measure the size is by taking the logarithm of the $BigInteger$ input.

Algorithm 4.3 The K2W polynomial multiplication.

1: **procedure** K2W(A, B) ▷ $Z = A \cdot B$
2: $n \leftarrow \max(\deg(A), \deg(B)) + 1$.
3: **if** $N \leq s$ **then** ▷ s is the size (maximum number of bits) of lookup table input
4: **return** $LUT(A, B)$.
5: **end if**
6: $n \leftarrow \lceil n/2 \rceil$.
7: $(A_1, A_0) \leftarrow SplitTwo(A)$.
8: $(B_1, B_0) \leftarrow SplitTwo(B)$.
9: $D_0 \leftarrow K2W(A_0, B_0)$. ▷ recursive productive calls
10: $D_1 \leftarrow K2W(A_1, B_1)$.
11: $D_{0,1} \leftarrow K2W(A_0 \oplus A_1, B_0 \oplus B_1)$.
12: $Z \leftarrow D_0 \oplus ((D_{0,1} \oplus D_0 \oplus D_1) \ll n) \oplus (D_1 \ll (n \ll 1))$. ▷ reconstruction
13: **return** Z.
14: **end procedure**

4.2.5 Lookup table of larger size

In Section 4.2.2, we have introduced our 3-bit look up table. However, larger look up tables are needed to make the algorithm faster. In order to have a look up table of an input size larger than 3 bits, we develop a method to generate a look up table which takes the number of bits of the inputs to the multiplication as an input. We use the base look up table and the Karatsuba algorithm presented in Section 4.2.4 to generate a larger one. Algorithm 4.4 shows the implementation of generating a larger look up table. After generating the new table $Table$, LUT is updated to have the new *larger* table.

4.2.6 The K3W algorithm

An extension of the K2W algorithm is the K3W algorithm. This is shown in Algorithm 4.5. In order to generate the splitting formula for the K3W algorithm, assume we have

Algorithm 4.4 Algorithm for generating a look up table.

```
1: procedure GenerateLookupTable(size)        ▷ size is the number of bits to the
   inputs of polynomial multiplication
2:     side ← 2^size.
3:     for a from 0 to side − 1 do
4:         for b from 0 to side − 1 do
5:             Table[a, b] ← Karatsuba(a, b).
6:         end for
7:     end for
8:     LUT ← Table.
9: end procedure
```

two polynomials $A(X)$ and $B(X)$ of degree $n-1$ in $\mathbb{F}_2[X]$. We can split these polynomials into three parts; assuming $Y = X^{n/3-1}$, then $A(Y) = A_0 + A_1Y + A_2Y^2$ and $B(Y) = B_0 + B_1Y + B_2Y^2$. Then, we evaluate the polynomials at $0, 1$ and ∞. However two other points are needed in the evaluation step to be able to generate the product. The evaluation Y modulo Y^2+Y+1 as suggested in [45] serves the purpose. The product calls are shown from Line 9 till Line 14 in Algorithm 4.5. This is followed by the reconstruction step.

4.2.7 The IK3W algorithm

In [7], a re-arrangement of the reconstruction process is proposed to save some computation. Algorithm 4.6 shows this improvement. The algorithm is similar to the previous one with the difference in the reconstruction step which starts at Line 15.

4.2.8 The B3W algorithm

Bernstein suggested a recursive algorithm for polynomial multiplication [18]. The idea is to do the evaluation at $0, 1, X, X+1$ and ∞. Algorithm 4.7 shows the Bernstein polynomial multiplication algorithm. The multi-evaluation starts at Line 9 and ends at Line 16. The reconstruction step includes division by $R_4 = \frac{R_3}{X^2+X}$. This can be done by first dividing R_3 by X and then dividing the result by $X+1$. The first one is a right shift shown at Line 26. The division by $X+1$ can be done by noticing that $R_4'[i] = R_4'[i+1] + R_4[i+2]$ where i is

Algorithm 4.5 K3W polynomial multiplication algorithm.

1: **procedure** K3W(A, B) ▷ $Z = A \cdot B$
2: $n \leftarrow \max(\deg(a), \deg(b))$.
3: **if** $N \leq s$ **then** ▷ s is the size (maximum number of bits) of lookup table input
4: **return** $LUT(A, B)$.
5: **end if**
6: $n \leftarrow \lceil n/3 \rceil$
7: $(A_2, A_1, A_0) \leftarrow SplitThree(A)$.
8: $(B_2, B_1, B_0) \leftarrow SplitThree(B)$.
9: $P_0 \leftarrow K3W(A_0, B_0)$. ▷ recursive product calls
10: $P_1 \leftarrow K3W(A_1, B_1)$.
11: $P_2 \leftarrow K3W(A_2, B_2)$.
12: $P_3 \leftarrow K3W(A_0 \oplus A_1, B_0 \oplus B_1)$.
13: $P_4 \leftarrow K3W(A_1 \oplus A_2, B_1 \oplus B_2)$.
14: $P_5 \leftarrow K3W(A_0 \oplus A_2, B_0 \oplus B_2)$.
15: $R_0 \leftarrow P_0 \oplus P_1$. ▷ reconstruction
16: $R_1 \leftarrow P_3 \oplus R_0$.
17: $R_2 \leftarrow P_1 \oplus P_0 \oplus P_2 \oplus P_5$.
18: $R_3 \leftarrow P_4 \oplus P_1 \oplus P_2$.
19: $Z \leftarrow P_0 \oplus (R_1 \ll n) \oplus (R_2 \ll (n \ll 1)) \oplus (R_3 \ll (3 \cdot n)) \oplus (P_2 \ll (n \ll 2))$
20: **return** Z.
21: **end procedure**

Algorithm 4.6 The IK3W polynomial multiplication.

1: **procedure** IK3W(A, B) ▷ $Z = A \cdot B$
2: $n \leftarrow \max(\deg(a), \deg(b))$.
3: **if** $N \leq s$ **then** ▷ s is the size (maximum number of bits) of lookup table input
4: **return** $LUT(A, B)$.
5: **end if**
6: $n \leftarrow \lceil n/3 \rceil$
7: $(A_2, A_1, A_0) \leftarrow SplitThree(A)$.
8: $(B_2, B_1, B_0) \leftarrow SplitThree(B)$.
9: $P_0 \leftarrow IK3W(A_0, B_0)$. ▷ recursive product calls
10: $P_1 \leftarrow IK3W(A_1, B_1)$.
11: $P_2 \leftarrow IK3W(A_2, B_2)$.
12: $P_3 \leftarrow IK3W(A_0 \oplus A_1, B_0 \oplus B_1)$.
13: $P_4 \leftarrow IK3W(A_1 \oplus A_2, B_1 \oplus B_2)$.
14: $P_5 \leftarrow IK3W(A_0 \oplus A_2, B_0 \oplus B_2)$.
15: $R_0 \leftarrow P_0 \oplus (P_1 \ll n) \oplus (P_2 \ll (n \ll 1))$. ▷ reconstruction
16: $R_1 \leftarrow R_0 \oplus (R_0 \ll n) \oplus (R_0 \ll (n \ll 1))$.
17: $Z \leftarrow R_1 \oplus (P_3 \ll n) \oplus (P_5 \ll (n \ll 1)) \oplus (P_4 \ll (3 \cdot n))$.
18: **return** Z.
19: **end procedure**

the bit index number. By assuming R_4 of degree n, then R_4 starts from $R_4[n-2] = R_3[n]$. These are done in the **while** loop at Line 28. The division makes the B3W algorithm slower compared to other algorithms.

4.2.9 The FE3W algorithm

Algorithms 4.8 and 4.9 show the FE3W polynomial multiplication algorithm. The idea is similar to the Bernstein multievaluation, but instead of evaluating at X and $X+1$, the field is extended to $\mathbb{F}_4$ and the polynomials are evaluated at $0, 1, \alpha, \alpha+1$ and ∞. Two of the product calls in Algorithm 4.8 invoke Algorithm 4.9 which is similar to the first algorithm. However, it takes the input in $\mathbb{F}_4$ format so the first two inputs are the first polynomial and the other two inputs constitute the second polynomial. Each polynomial is represented by two inputs: one is the non-α part and the other is the α-part.

In Algorithm 4.9, the way to access the look-up table is done by three accesses to the table. Assume that we have $a_0 + a_1\alpha$ and $b_0 + b_1\alpha$, then their product is $a_0b_0 + a_0b_1\alpha + a_1b_0\alpha + a_1b_1\alpha^2$. But $\alpha^2 = \alpha + 1$, hence the result can be rewritten as $(a_0b_0 + a_1b_1) + (a_0b_1 + a_1b_0 + a_1b_1)\alpha$. However, this needs four accesses to the look-up table. In order to reduce the number of accesses, we utilize the fact that $(a_0 + a_1)(b_0 + b_1) + a_0b_0$ is equal to the α part. This reduces the number of look-up table accesses to three. This starts at Line 4 and ends at Line 7 in Algorithm 4.9.

4.3 Reduction Algorithms

In section 2.2.4, we have introduced reduction and discussed fast reduction algorithms. In this section, we show our implementation of reduction algorithms. Algorithms 4.10 show the reduction algorithm where the polynomial has maximum size $m = 163$ module the NIST recommended field $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$. For other NIST recommended fields refer to algorithms A.1, A.2, A.3, and A.4 in Appendix A. These algorithms take advantage of the knowledge of the field and the small number of non-zeros in the field defining polynomials (trinomial or pentanomial).

Algorithm 4.7 B3W polynomial multiplication algorithm.

1: **procedure** B3W(A, B) ▷ $Z = A \cdot B$
2: $n \leftarrow \max(\deg(A), \deg(B))$.
3: **if** $N \leq s$ **then** ▷ s is the size (maximum number of bits) of lookup table input
4: **return** $LUT(A, B)$.
5: **end if**
6: $n \leftarrow \lceil n/3 \rceil$.
7: $(A_2, A_1, A_0) \leftarrow SplitThree(A)$.
8: $(B_2, B_1, B_0) \leftarrow SplitThree(B)$.
9: $M_1 \leftarrow A_0 \oplus A_1$. ▷ multi-evaluation
10: $M_1^{'} \leftarrow B_0 \oplus B_1$.
11: $M_2 \leftarrow A_1 \oplus A_2$.
12: $M_2^{'} \leftarrow B_1 \oplus B_2$.
13: $M_3 \leftarrow A_0 \oplus M_2$.
14: $M_3^{'} \leftarrow B_0 \oplus M_2^{'}$.
15: $M_4 \leftarrow M_1 \oplus M_2$.
16: $M_4^{'} \leftarrow M_1^{'} \oplus M_2^{'}$.
17: $P_0 \leftarrow B3W(A_0, B_0)$. ▷ recursive product calls
18: $P_1 \leftarrow B3W(M_1, M_1^{'})$.
19: $P_2 \leftarrow B3W(M_3, M_3^{'})$.
20: $P_3 \leftarrow B3W(M_4, M_4^{'})$.
21: $P_4 \leftarrow B3W(A_2, B_2)$.
22: $R_0 \leftarrow P_2 \oplus P_3$. ▷ reconstruction
23: $R_1 \leftarrow P_0 \oplus ((P_0 \oplus P_1) \ll n)$.
24: $R_2 \leftarrow P_2 \oplus ((R_0 \ll n) \oplus (R_0 \ll 1))$.
25: $R_3 \leftarrow R_1 \oplus R_2 \oplus ((P_4 \ll 4) \oplus (P_4 \ll 1))$.
26: $R_4 \leftarrow R_3 \gg 2$. ▷ Start division $R_4 \leftarrow \frac{R_3}{X^2+X}$
27: $R_4^{'} \leftarrow R_4$
28: **while** $R_4^{'} \neq 0$ **do**
29: $R_4^{'} \leftarrow R_4^{'} \gg 1$.
30: $R_4 \leftarrow R_4 \oplus R_4^{'}$.
31: **end while** ▷ End division
32: $R_5 \leftarrow (R_4 \ll (n \ll 1)) \oplus (R_4 \ll n)$.
33: $Z \leftarrow R_1 \oplus (P_4 \ll (n \ll 2)) \oplus (P_4 \ll n) \oplus R_5$.
34: **return** Z.
35: **end procedure**

Algorithm 4.8 The FE3W polynomial multiplication algorithm.

procedure FE3W(A, B) ▷ $Z = A \cdot B$
  $n \leftarrow \max(\deg(A), \deg(B))$.
  if $N \leq s$ **then** ▷ s is the size (maximum number of bits) of lookup table input
    return $LUT(A, B)$.
  end if
  $n \leftarrow \lceil n/3 \rceil$.
  $(A_2, A_1, A_0) \leftarrow SplitThree(A)$.
  $(B_2, B_1, B_0) \leftarrow SplitThree(B)$.
  $M_1 \leftarrow A_0 \oplus A_1,\ M_1^{'} \leftarrow B_0 \oplus B_1$. ▷ multi-evaluation
  $M_2 \leftarrow A_1 \oplus A_2,\ M_2^{'} \leftarrow B_1 \oplus B_2$.
  $M_{3\alpha} \leftarrow M_2,\ M_{3\alpha}^{'} \leftarrow M_2^{'}$.
  $M_4 \leftarrow M_1,\ M_{4\alpha} \leftarrow M_{3\alpha}$.
  $M_4^{'} \leftarrow M_1^{'},\ M_{4\alpha}^{'} \leftarrow M_{3\alpha}^{'}$.
  $M_5 \leftarrow M_4 \oplus M_2,\ M_{5\alpha} \leftarrow M_{4\alpha}$.
  $M_5^{'} \leftarrow M_4^{'} \oplus M_2^{'},\ M_{5\alpha}^{'} \leftarrow M_{4\alpha}^{'}$.
  $M_6 \leftarrow M_1 \oplus A_2,\ M_6^{'} \leftarrow M_1^{'} \oplus B_2$.
  $P_0 \leftarrow FE3W(A_0, B_0)$. ▷ recursive product calls
  $P_1 \leftarrow FE3W(M_6, M_6^{'})$.
  $P_2 \leftarrow FE3W\mathbb{F}_4(M_5, M_{5\alpha}, M_5^{'}, M_{5\alpha}^{'})$.
  $P_3 \leftarrow FE3W\mathbb{F}_4(M_4, M_{4\alpha}, M_4^{'}, M_{4\alpha}^{'})$.
  $P_4 \leftarrow FE3W(A_2, B_2)$.
  $R_0 \leftarrow P_2[0] \oplus P_3[0]$. ▷ reconstruction
  $R_{0\alpha} \leftarrow P_2[1] \oplus P_3[1]$
  $R_1 \leftarrow R_{0\alpha}$.
  $R_2 \leftarrow R_0 \oplus R_{0\alpha}$.
  $R_3 \leftarrow P_1 \oplus R_2$.
  $R_4 \leftarrow (R_3 \ll n) \oplus (R_3 \ll (n \ll 1)) \oplus (R_3 \ll (3 \cdot n))$
  $R_5 \leftarrow P_0 \oplus (P_4 \ll n)$.
  $R_6 \leftarrow R_5 \oplus (R_5 \ll (3 \cdot n))$.
  $Z \leftarrow R_6 \oplus R_4 \oplus (R_1 \ll (3 \cdot n)) \oplus (P_2[0] \ll (n \ll 1)) \oplus (P_3[0] \ll n)$.
  return Z.
end procedure

Algorithm 4.9 FE3W in $\mathbb{F}_{2^2}$ polynomial multiplication algorithm.

1: **procedure** FE3W$\mathbb{F}_4(A, A_\alpha, B, B_\alpha)$ ▷ $Z = A \cdot B$
2: $n \leftarrow \max(\deg(A), \deg(B))$.
3: $n_\alpha \leftarrow \max(\deg(A_\alpha), \deg(B_\alpha))$.
4: **if** $n \leq s$ **then** ▷ s is the size (maximum number of bits) of lookup table input.
5: $C \leftarrow LUT(A, B)$, $D \leftarrow LUT(A_\alpha, B_\alpha)$.
6: $Z[0] \leftarrow C \oplus D$, $Z[1] \leftarrow LUT(A \oplus A_\alpha, B \oplus B_\alpha) \oplus C$.
7: **return** Z. .
8: **end if**
9: $n \leftarrow \lceil n/3 \rceil$.
10: $(A_2, A_1, A_0) \leftarrow SplitThree(A)$, $(A_{2\alpha}, A_{1\alpha}, A_{0\alpha}) \leftarrow SplitThree(A_\alpha)$.
11: $(B_2, B_1, B_0) \leftarrow SplitThree(B)$, $(B_{2\alpha}, B_{1\alpha}, B_{0\alpha}) \leftarrow SplitThree(B_\alpha)$.
12: $M_1 \leftarrow A_0 \oplus A_1$, $M_{1\alpha} \leftarrow A_{0\alpha} \oplus A_{1\alpha}$,$M_1^{'} \leftarrow B_0 \oplus B_1$, $M_{2\alpha}^{'} \leftarrow B_{0\alpha}B_{1\alpha}$.
13: $M_2 \leftarrow A_1 \oplus A_2$, $M_{2\alpha} \leftarrow A_{1\alpha} \oplus A_{2\alpha}$, $M_2^{'} \leftarrow B_1 \oplus B_2$, $M_{2\alpha}^{'} \leftarrow B_{1\alpha} \oplus B_{2\alpha}$.
14: $M_3 \leftarrow M_{2\alpha}$, $M_{3\alpha} \leftarrow M_2 \oplus M_{2\alpha}$, $M_{3\alpha}^{'} \leftarrow M_2^{'}$, $M_{3\alpha}^{'} \leftarrow M_2^{'}$.
15: $M_4 \leftarrow M_1 \oplus M_3$, $M_{4\alpha} \leftarrow M_{3\alpha}\oplus$.
16: $M_4 \leftarrow M_1 \oplus M_3$, $M_{4\alpha} \oplus M_{1\alpha} \oplus M_{3\alpha}$,$M_4^{'} \leftarrow M_1^{'} \oplus M_3^{'}$, $M_{4\alpha}^{'} \leftarrow M_{3\alpha}^{'} \oplus M^{'}$.
17: $M_5 \leftarrow M_4 \oplus M_2$, $M_{5\alpha} \leftarrow M_{4\alpha} \oplus M_{2\alpha}$, $M_5^{'} \leftarrow M_4^{'} \oplus M_2^{'}$, $M_{5\alpha}^{'} \leftarrow M_{4\alpha}^{'} \oplus M_{2\alpha}^{'}$.
18: $M_6 \leftarrow M_1 \oplus A_2$, $M_{6\alpha} \leftarrow M_{1\alpha} \oplus A_{2\alpha}$, $M_6^{'} \leftarrow M_1^{'} \oplus B_2$, $M_{6\alpha}^{'} \leftarrow M_{1\alpha}^{'} \oplus B_{2\alpha}$
19: $P_0 \leftarrow FE3W\mathbb{F}_4(A_0, A_{0\alpha}, B_0, B_{0\alpha})$. ▷ recursive product calls
20: $P_1 \leftarrow FE3W\mathbb{F}_4(M_6, M_{6\alpha}, M_6^{'}), M_{6\alpha}^{'})$.
21: $P_2 \leftarrow FE3W\mathbb{F}_4(M_5, M_{\alpha_5}, M_5^{'}, M_{\alpha_5}^{'})$.
22: $P_3 \leftarrow FE3W\mathbb{F}_4(M_4, M_{\alpha_4}, M_4^{'}, M_{\alpha_4}^{'})$.
23: $P_4 \leftarrow FE3W\mathbb{F}_4(A_2, A_{2\alpha}, B_2, B_{2\alpha})$.
24: $R_0 \leftarrow P_2[0] \oplus P_3[0]$. ▷ reconstruction
25: $R_{\alpha_0} \leftarrow P_2[1] \oplus P_3[1]$.
26: $R_1 \leftarrow R_{\alpha_0}$, .
27: $R_2 \leftarrow R_0 \oplus R_{\alpha_0}$.
28: $R_3 \leftarrow P_1 \oplus R_2$.
29: $R_4 \leftarrow (R_3 \ll n) \oplus (R_3 \ll (n \ll 1)) \oplus (R_3 \ll (3 \cdot n))$.
30: $R_5 \leftarrow P_0 \oplus (P_4 \ll n)$.
31: $R_6 \leftarrow R_5 \oplus (R_5 \ll (3 \cdot n))$.
32: $Z \leftarrow R_6 \oplus R_4 \oplus (R_1 \ll (3 \cdot n)) \oplus (P_2[0] \ll (n \ll 1)) \oplus (P_3[0] \ll n)$.
33: **return** Z.
34: **end procedure**

Algorithm 4.10 Reduction modulo $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$.

1: **procedure** REDUCE163(x) ▷ x is a polynomial of size maximum $2m - 2$ where $m = 163$.
2: $ones \leftarrow 2^{163} - 1$.
3: $temp \leftarrow x \gg 163$.
4: $result \leftarrow x \ \& \ ones$
5: $result \leftarrow result \oplus (temp \ll 7) \oplus (temp \ll 6) \oplus (temp \ll 3) \oplus temp$.
6: $temp \leftarrow result \gg 163$.
7: $result \leftarrow result \oplus (temp \ll 7) \oplus (temp \ll 6) \oplus (temp \ll 3) \oplus temp$.
8: $result \leftarrow result \ \& \ ones$.
9: **return** $result$.
10: **end procedure**

4.4 Elliptic Curve Algorithms

An elliptic curve point multiplication uses point doubling and point addition algorithms. As a result, improvements in these algorithms lead to higher speed in point multiplication algorithm [6]. In section 2.3.5, we have introduced point addition and point doubling algorithms. However, in [6] an improvement has been proposed. In this section, these proposed point addition and point doubling algorithms are presented.

4.4.1 Point addition

Algorithm 4.11 shows the CANH point addition algorithm. This method calls a modification of the method FieldExtension$\mathbb{F}_4$ such that it takes the first input in $\mathbb{F}_2$ and the other in $\mathbb{F}_4$. We refer this method in the code as $Multiply_{2,4}$. This is also useful when there is simultaneous occurrence of field multiplication like AB and AC. The result of both multiplications can be achieved by a single call to the $Multiply_{2,4}$ method. Any other multiplication is just denoted by " $\cdot$ ", which can be any of the polynomial multiplication methods explained earlier. A call of an appropriate reduction algorithm has to be understood after any call of polynomial multiplication or square methods.

Algorithm 4.11 CANH point addition algorithm.

1: **procedure** CANHPointAdd(P, Q) ▷ P and Q are two points on a curve
2: **if** $P = \infty$ **then**
3: **return** Q.
4: **end if**
5: **if** $Q = \infty$ **then**
6: **return** P.
7: **end if**
8: $T_1 \leftarrow Multiply_{2,4}(Z_Q, Y_P, X_P)$.
9: $T_2 \leftarrow T_1[0]$, $T_3 \leftarrow T_1[1]$.
10: $T_4 \leftarrow Multiply_{2,4}(Z_P, Y_Q, X_Q)$.
11: $T_5 \leftarrow T_4[0]$, $T_6 \leftarrow T_4[1]$.
12: $T_7 \leftarrow T_2 \oplus T_5$.
13: $T_8 \leftarrow T_3 \oplus T_6$.
14: $T_9 \leftarrow T_7 \oplus T_8$.
15: $T_{10} \leftarrow {T_8}^2$. ▷ Calling Square method
16: $T_{11} \leftarrow Z_P \cdot Z_Q$.
17: $T_{12} \leftarrow T_8 \cdot T_{10}$.
18: $T_{13} \leftarrow Multiply_{2,4}(T_7, T_9, T_3)$.
19: $T_{14} \leftarrow T_{13}[0]$.
20: $T_{15} \leftarrow T_{13}[1]$.
21: $T_{16} \leftarrow T_{14} \oplus (a_2, T_{10})$. ▷ a_2 is the value in the Weierstrass equation.
22: $T_{17} \leftarrow Multiply_{2,4}(T_{20}, T_8, T_9)$.
23: $T_{18} \leftarrow T_{17}[0]$.
24: $T_{19} \leftarrow T_{17}[1]$.
25: $Z_R \leftarrow T_{19}$.
26: $T_{20} \leftarrow T_{18} \oplus T_{12}$.
27: $T_{21} \leftarrow Multiply_{2,4}(T_{20}, T_8, T_9)$.
28: $T_{22} \leftarrow T_{21}[0]$.
29: $T_{23} \leftarrow T_{21}[1]$.
30: $X_R \leftarrow T_{22}$.
31: $Y_R \leftarrow (T_{10} \cdot (T_{15} \oplus (T_8 \cdot T_2))) \oplus T_{23}$.
32: **return** R.
33: **end procedure**

4.4.2 Point doubling

Algorithm 4.12 shows the CANH point doubling algorithm. Similar techniques to point addition algorithm are used in this point doubling algorithm to achieve a higher speed.

Algorithm 4.12 CANH point doubling algorithm.

1: **procedure** CANHPOINTDOUBLE(P) ▷ P is a point on a curve
2: **if** $P = \infty$ **then**
3: **return** P.
4: **end if**
5: $T_1 \leftarrow {X_p}^2$. ▷ Calling Square method
6: $T_2 \leftarrow Multiply_{2,4}(Z_p)$.
7: $T_3 \leftarrow T_2[0]$.
8: $T_4 \leftarrow T_2[1]$.
9: $T_5 \leftarrow T_1 \oplus T_3$.
10: $T_6 \leftarrow T_4 \oplus T_5$.
11: $T_7 \leftarrow {T_4}^2$.
12: $T_8 \leftarrow (T_5 \cdot T_6) \oplus (a_2 \cdot T_7)$.
13: $T_9 \leftarrow Multiply_{2,4}(T_8, T_4, T_6)$.
14: $T_{10} \leftarrow T_9[0]$.
15: $T_{11} \leftarrow T_9[1]$.
16: $T_{12} \leftarrow {T_1}^2$.
17: $T_{13} \leftarrow Multiply_{2,4}(T_4, T_{12}, T_7)$.
18: $T_{14} \leftarrow T_{13}[0]$.
19: $T_{15} \leftarrow T_{13}[1]$.
20: $X_R \leftarrow T_{10}$.
21: $Y_R \leftarrow T_{11} \oplus T_{14}$.
22: $Z_R \leftarrow T_{15}$.
23: **return** R.
24: **end procedure**

4.5 Summary

Implementation oriented view of algorithms of our interest have been given in this chapter. This view helps us look at the points that need further improvement for subsequent implementation versions. Each algorithm has been implemented in a class and each set of related algorithms has their interface. Examples include polynomial multiplication algorithms and reduction algorithms. The K2W, K3W, IK3W, B3W and FE3W algorithms have been considered under the set of polynomial multiplication algorithms. We have shown the implementation details of using the field extension method $Multiply_{2,4}$. The following chapter has the results of our implementation.

Chapter 5

Timing Results

In this chapter, we discuss the timing results of multiplication of polynomial of different sizes. Then, we discuss the timing results of point multiplication algorithms.

5.1 Machine Information

Our software ran on Intel Core i7 with 2.8 GHz and 8.00 GB memory (RAM). The operating system is Windows 7 Professional. As mentioned earlier, the programming language used is C# with .Net framework 4.0, and the development environment is Visual Studio 2010.

5.2 Polynomial Multiplication Timing

In this section, timing results for polynomial multiplication are presented.

5.2.1 Experiment setup

The goal is to determine how long it will take for a specific polynomial multiplication algorithm to multiply two polynomials of different sizes. The length n, which is $d+1$ where d is the degree, ranges from 8 bits to 1200 bits at an increment of 8. Each multiplication is run 150 times and their timings are averaged. The experiment is run for the K2W,

K3W, IK3W, B3W, and FE3W algorithms. Also these algorithms are run with different padding methods: power and multiple. In the power padding, the length of the input is incremented such that it is a power of 2 in the case of K2W, and a power of 3 in the case of other polynomial multiplication methods. In the multiple padding, a number of zeros is padded to the input such that its new length is multiple of 3; this needs to be done in every iteration to be able to divide the polynomial into three equal sub-polynomials.

5.2.2 Timing results

Figure 5.1 shows the number of clock cycles it takes for each multiplication algorithm with different sizes of polynomials in the case of multiple padding.

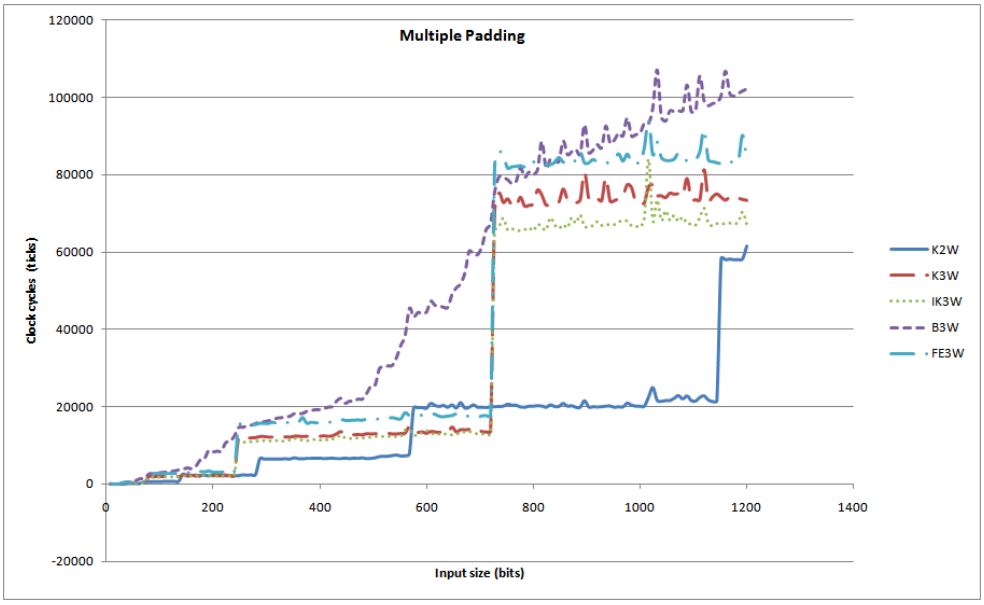


Figure 5.1: The number of clock cycles required for each polynomial multiplication algorithm at different input sizes in the case of multiple padding.

From Figure 5.1, we notice that there are points where there is a noticeable increase in the time or *jump* needed to complete the multiplication compared to previous sizes. In the case of the K2W algorithm, the sizes are close to $128, 256, 512$, and 1024, which are all actually a power of two. The reason is that, at these points, an additional round

of recursive multiplication call is needed. Similarly, other algorithms have jumps at sizes 81, 243, and 729, because they are power of 3 and are used in 3-way split algorithms. The K2W algorithm seems to perform best except in the range from 512 to 729 where the IK3W is the best. The B3W algorithm takes longer than other algorithms almost in all cases in our experiments. The jumps are not very clear in the case of the B3W algorithm because two of the recursive calls are longer than the length divided by 3. The B3W algorithm takes longer in general because of the division it requires. We note that the jumps can be turned into gradual increases by carefully optimizing computations involved in each recursion. In this work, we have not tried such optimizations for ease of implementation.

Figure 5.2 shows the number of clock cycles needed for multiplication of two polynomials of different sizes where the power padding method is used.

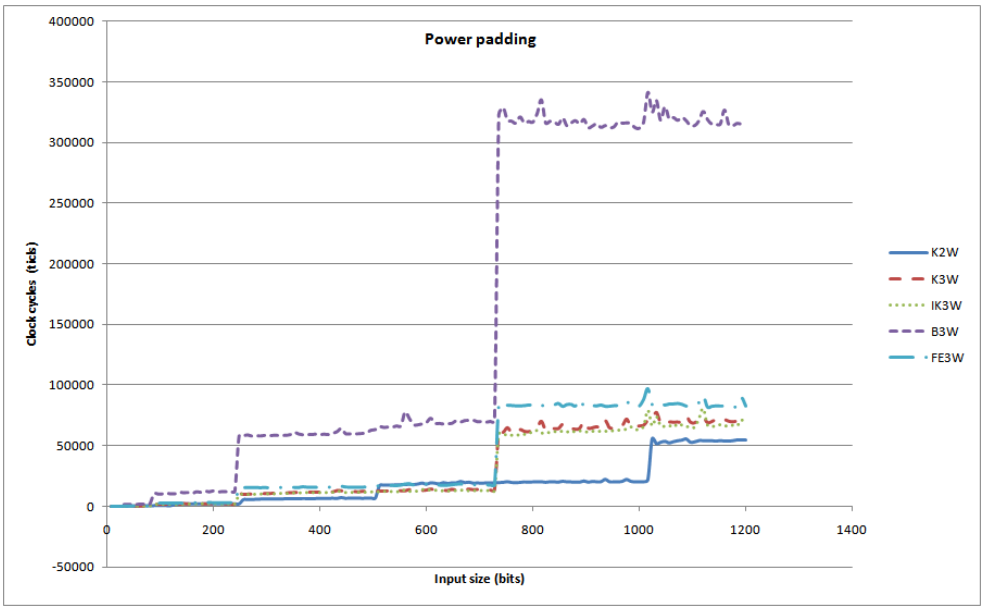


Figure 5.2: The number of clock cycles spent for each polynomial multiplication algorithm at different input sizes in the case of power padding.

It is clear from Figure 5.2 that the jumps are sharper in the case of power padding than in the multiple padding because of the way power and multiple are different. Locations of the jumps are similar to the case of multiple at powers of 2 and 3.

Since the code give us the correct result and it strongly match the algorithm, this gives

us strong indication of the correctness of the timing results. The staircase look of the figure is due to the nature of split location and the recursive implementation used in polynomial multiplication. The figure also covers a very large range of the field size, a huge jump at the end.

In the K2W algorithm, it is observed that power padding jumps occur before multiple padding jumps. In most of the sizes, the number of clock cycles (ticks) is close in both cases. In IK3W algorithm, until about 700 bits, multiple and power are almost the same. Power achieves better results after about 715 bits. In the B3W algorithm, Multiple padding performs much better than the power padding. Two recursive calls of the B3W algorithm have some extra bits in addition to the previous size divided by three making the jump higher in the case of power. The jumps in multiple are not very clear, but one can notice a slow increase.

5.3 Polynomial Multiplication for NIST Field Sizes

The point multiplication algorithm in elliptic curve cryptography must be in a field. Here we consider the five binary fields corresponding to NIST recommended curves B163, B233, B283, B409, and B571 [9]. In this section, we show the result of polynomial multiplication for these fields. The proposed point multiplication uses the $Multiply_{2,4}$ method, that is a modification of the 3-way field extension method that takes one input in $\mathbb{F}_2$ and the other in $\mathbb{F}_4$. Therefore, this polynomial multiplication method is also implemented and tested.

5.3.1 Experiment setup

The input size is chosen to be equal to the field and the number of times each multiplication is performed is 150 so that we can determine its average. The polynomial multiplication algorithms tested are the K2W, the K3W, the IK3W, the FE3W, the $Multiply_{2,4}$, which is a modification of FE3W, and the B3W algorithms.

5.3.2 Timing results

Figure 5.3 shows the number of clock cycles for polynomial multiplication algorithms in the case of multiple padding. It can be observed that the larger the field, the more number

of clock cycles needed. Moreover, the K2W algorithm performs the best, followed by the IK3W, then the K3W algorithm, after that the FE3W algorithm, and then the $Multiply_{2,4}$, and finally the B3W algorithm. In the fields $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$, the IK3W algorithm performs better than the K2W.

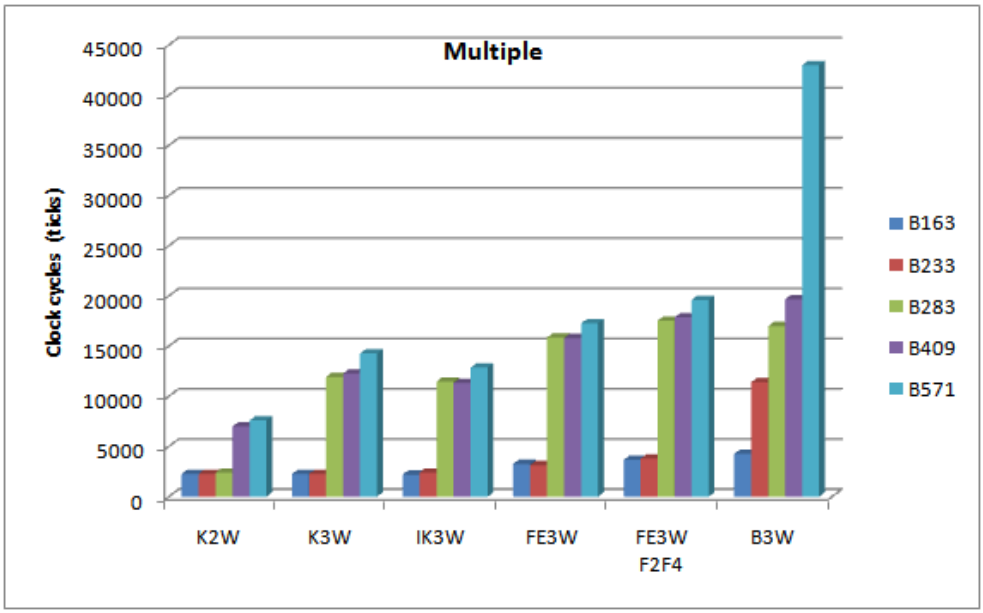


Figure 5.3: Polynomial multiplication clock cycles at NIST fields.

5.4 Point Multiplication Results

In this section, the timing results of point multiplication are shown. The focus is to see how the CANH algorithms improves over the conventional algorithms in point multiplication. The CANH algorithms is the one that makes use of $Multiply_{2,4}$ or FE3W-F2F4 algorithm.

5.4.1 Experiment setup

Three types of polynomial multiplication algorithms are used. These are the K2W [14], the IK3W [7], and the B3W [18]. Also we use several point multiplication algorithms: the

Table 5.1: The legend table to be used in the tables of results of point multiplication algorithm.

	Legend	Meaning
Polynomial Multiplication	A	K2W Algorithm [14]
	B	IK3W Algorithm [7]
	C	B3W Algorithm [18]
Point Multiplication	I	double-and-add algorithm (Page 96 of [8])
	II	Binary NAF algorithm (Page 99 of [8])
	III	Window NAF algorithm (Page 100 of [8])
Point addition/ doubling	Conventional	Reported in [12]
	CANH	The work by Cenk M. et al. [6]
Padding Method	M	Multiple padding
	P	Power padding

general double-and-add algorithm (Page 96 of [8]), the binary NAF algorithm (Page 99 of [8]), and the window NAF algorithm (Page 100 of [8]). Both the conventional [12] and the CANH [6] point addition and point doubling algorithms that are invoked by the point multiplication algorithm are used. We also use two types of padding methods: power and multiple. Table 5.1 shows the legends of these algorithms used in the result tables that follow. Each point multiplication algorithm is run 100 times for every possible case and then their average is computed.

5.4.2 Timing results

In Table 5.2, the timing results of point multiplication is shown for B163. In each case, there is an improvement when using the CANH algorithm: The greatest improvement is when using the B3W algorithm, the least improvement result from using the IK3W algorithm. When using the power padding in case of the B3W algorithm, we can see that the improvement of the CANH method is quite significant, because of the longer time the B3W algorithm needs to perform polynomial multiplication in the case of power padding.

Tables 5.3, 5.4, 5.5, and 5.6 show the results of point multiplication algorithms in the cases of the fields B233, B283, B409, and B571 accordingly. The use of the CANH algorithm improves the overall performance in most cases. When using the K2W algorithm

and the field is not B233, there is a reduction of the speed because of the efficiency of the K2W algorithm in these cases. The greatest improvement is generally found when using the B3W algorithm.

In these tables, the **bold** font is used when we have the fastest configuration combinations. In the case of B163, the minimum number of ticks is 3645008, which occurs when using the CANH point addition/doubling algorithms, the IK3W polynomial multiplication method, Window NAF point multiplication and multiple padding. Similar configuration for the minimum number of ticks required in the case of B233 but the power padding method is used. In the other cases, i.e. B283, B409, and B571, the best configuration is to use the K2W polynomial multiplication algorithm, Window NAF point multiplication, and conventional point addition/doubling algorithms.

5.5 Summary

We have a number of experiments in order to get our timing results. We have seen how different polynomial algorithms work under different sizes and different configurations. The K2W, K3W, IK3W, B3W, and FE3W algorithms have been considered. We have run several experiments for point multiplication to see the effect of using the CANH algorithm and see what configuration gives the best results. We have found that the use of the CANH point addition/doubling with the IK3W algorithms work the best in the cases of B163 and B233. However, in other cases, the K2W algorithm with the conventional point addition/doubling algorithms gives the minimum number of clock cycles. We have also found that the use of the CANH algorithms gives us better results than the conventional algorithms, except for the K2W algorithm for B283, B409 and B571.

Table 5.2: Point multiplication clock cycles in the case of B163.

Poly. Mult.	Point Mult	Padding	Clock cycles		Improvement %
			Conventional	CANH	
A	I	M	5421410	4827576	10.95
		P	5381407	4785173	11.08
	II	M	4569161	4030330	11.79
		P	4679167	4212340	9.98
	III	M	4049031	3941025	2.67
		P	4212840	4093934	2.82
B	I	M	4966284	4695868	5.45
		P	5115592	4773573	6.69
	II	M	4238042	3939225	7.05
		P	4449254	4082233	8.25
	III	M	3740613	**3645008**	2.56
		P	3968727	3903223	1.65
C	I	M	9544345	5754029	39.71
		P	27213956	10193883	62.54
	II	M	8079862	4829476	40.23
		P	22838306	8761101	61.64
	III	M	7224513	6512172	9.86
		P	20269759	16894066	16.65

Table 5.3: Point multiplication clock cycles in the case of B233.

Poly. Mult.	Point Mult	Padding	Clock cycles		Improvement %
			Conventional	CANH	
A	I	M	8067661	7285916	9.69
		P	8154766	7331719	10.09
	II	M	6845391	6148751	10.18
		P	6947997	6363763	8.41
	III	M	6004543	5853834	2.51
		P	6172353	5934839	3.85
B	I	M	7694440	7160609	6.94
		P	7551031	7019201	7.04
	II	M	6556475	6070547	7.41
		P	6422567	6016144	6.33
	III	M	5749328	5640322	1.9
		P	5744828	**5633122**	1.94
C	I	M	37935669	14594034	61.53
		P	42462028	16006315	62.3
	II	M	32267545	13665581	57.65
		P	35800247	13105349	63.39
	III	M	30864865	24677311	20.05
		P	31870722	26231500	17.69

Table 5.4: Point multiplication clock cycles in the case of B283.

Poly. Mult.	Point Mult	Padding	Clock cycles		Improvement %
			Conventional	CANH	
A	I	M	10881222	30535046	-180.62
		P	25665667	33903939	-32.1
	II	M	9355335	26849535	-187
		P	21625936	29359079	-35.76
	III	M	**8373878**	12218498	-45.91
		P	18937283	20858993	-10.15
B	I	M	49123109	41300262	15.92
		P	43775203	39204142	10.44
	II	M	41800490	35018402	16.22
		P	37235629	32846278	11.79
	III	M	36094264	34593378	4.16
		P	32238743	31256387	3.05
C	I	M	67240145	45650311	32.11
		P	237361776	90522477	61.86
	II	M	57641296	38369894	33.43
		P	206409505	76017947	63.17
	III	M	50600694	46111737	8.87
		P	180311613	148273180	17.77

Table 5.5: Point multiplication clock cycles in the case of B409.

Poly. Mult.	Point Mult	Padding	Clock cycles		Improvement %
			Conventional	CANH	
A	I	M	42828349	53475558	-24.86
		P	40238401	50129267	-24.58
	II	M	36159768	45116980	-24.77
		P	34329963	42545533	-23.93
	III	M	31501001	33538918	-6.47
		P	**29895409**	32053533	-7.22
B	I	M	73310193	61310006	16.37
		P	70965659	60355652	14.95
	II	M	62334065	51856466	16.81
		P	59840922	49699942	16.95
	III	M	53819278	50199171	6.73
		P	52191385	50034261	4.13
C	I	M	113696803	68033891	40.16
		P	352578366	130656273	62.94
	II	M	96625126	57558692	40.43
		P	297647624	107657457	63.83
	III	M	86156327	77154112	10.45
		P	262148494	221850089	15.37

Table 5.6: Point multiplication clock cycles in the case of B571.

Poly. Mult.	Point Mult	Padding	Clock cycles		Improvement %
			Conventional	CANH	
A	I	M	62667584	76953201	-22.8
		P	158105943	107627655	31.93
	II	M	52940027	65044520	-22.86
		P	134841912	90844195	32.63
	III	M	**45986630**	48924698	-6.39
		P	123196646	108932830	11.58
B	I	M	107382341	89085095	17.04
		P	109359555	91792750	16.06
	II	M	90293664	74386454	17.62
		P	92784406	79228031	14.61
	III	M	78284477	75024891	4.16
		P	83380069	79483946	4.67
C	I	M	347784092	148922117	57.18
		P	559202784	203432235	63.62
	II	M	294669654	122611913	58.39
		P	477184193	175774853	63.16
	III	M	258508485	221163849	14.45
		P	416272509	337970130	18.81

Chapter 6

CANH Method on Pairing and Hyperelliptic Curves

In this chapter, we use the CANH trick in order to improve the performance of the pairing computation algorithm and the major operations of hyperelliptic, i.e. divisor addition and doubling algorithms. Hence, this chapter has essentially two parts: the first part concentrates on pairing and the second one focuses on hyperelliptic curves.

In the pairing part covered in Section 6.2, we propose a new methodology to speed up the Eta pairing computation. To improve elliptic curve point addition and doubling algorithms, [6] makes use of the presence of a common factor between multiplications. We notice the existence of multiplications having common factor, inside the Miller loop of the Eta pairing algorithm as in [46]. In this work, we present explicit formulæ for the Miller loop computation that take advantage of the common factors. We also analyze the cost of the proposed formulæ and compare it with that of the traditional formulae.

In the hyperelliptic curve part covered in Section 6.3, since CANH trick [6] utilizes the presence of common factors in polynomial multiplications to reduce the time needed for doing polynomial multiplications, we apply this trick over the explicit formulæ of the group operations of hyperelliptic curves of genus 2. To our knowledge, the work of Wollinger and Kovtun [47] presents the fastest explicit formulæ over even characteristic genus 2 hyperelliptic curves. We utilize the presence of common factors in the formulæ to apply the CANH trick to it. The outcome of the research shows an improvement when field size is higher than a certain threshold. The improvement increases for higher field sizes.

Before going to the details, the notations used and their corresponding complexity is given in Section 6.1. At the end of this chapter, a summary of our findings is presented in Section 6.4.

6.1 Notations

In Section 3.1.4 we discussed the 3-way split algorithm proposed in [7]. This polynomial multiplication is denoted as $\mathbf{M}_2$, where the number of additions is $5.27n^{\log_3(6)} - 6.67n + 1.4$ and the number of multiplications is $n^{\log_3(6)}$; therefore, the total arithmetic complexity is $6.27n^{\log_3(6)} - 6.67n + 1.4$. A 3-way split polynomial multiplications with five multiplications based on field extension is described [7] and presented in Section 3.1.6. The number of additions of this method is $27.75n^{\log_3(5)} - 9.67n\log_3(n) - 28.5n + 0.75$ and the number of multiplications is $3n^{\log_3(5)} - 2n$. The total arithmetic complexity is: $30.75n^{\log_3(5)} - 9.67n\log_3(n) - 30.5n + 0.75$. In Section 3.1.7 we discussed the reduction of two $\mathbb{F}_2[X]$ multiplication to one $\mathbb{F}_4[X]$ multiplication [6]. For this a field extension based multiplication algorithm is used where one input in $\mathbb{F}_2[X]$ and the other is in $\mathbb{F}_4[X]$; it is denoted as $\mathbf{M}_{2,4}$. In this case, the number of additions is $26.75n^{\log_3(5)} - 2.33n\log_3(n) - 32n + 10.5$ and the number of multiplications is $4n^{\log_3(5)} - 2n$, therefore the total arithmetic complexity is: $30.75n^{\log_3(5)} - 2.33n\log_3(n) - 34n + 10.5$.

Table 6.1 summarizes the notations used in this chapter to denote a certain operation cost and the total arithmetic complexity that certain operation costs. The first one is the cost of adding two polynomials of degree $n-1$; it is denoted by **Add**, and therefore the arithmetic cost is n. The second one is the cost of squaring, since $\left(\sum_{i=0}^{n-1} a_i x^i\right)^2 = \sum_{i=0}^{n-1} a_i x^{2i}$ so it is computation free. Since we need to perform reduction after that we choose to include the cost of reduction cost of squaring. In [48] it is proven that the reduction ($\mathbf{R}_2$) be done in $(r-1)(n-1)$ bit additions where r is the Hamming weight of the irreducible polynomial. Since in practice there is always a pentanomial of degree n, the reduction can be done at $4(n-1)$ bit additions. Reduction in $\mathbb{F}_4$, denoted as $\mathbf{R}_4$, requires double the cost i.e. $8(n-1)$ bit operations. The last one is the cost of square root denoted by **sqrt** and is used only in pairing; this cost choice is explained in Section 6.2.3.

Table 6.1: Summary of operations and their complexity costs.

Operation description	Symbol	Cost
Addition in $\mathbb{F}_2[X]$	**Add**	n
Squaring in $\mathbb{F}_2[X]$	**S**	$4n-4$
Reduction in $\mathbb{F}_2[X]$	$\mathbf{R}_2$	$4n-4$ (pentanomial), $2n-2$ (trinomial)
Reduction in $\mathbb{F}_4[X]$	$\mathbf{R}_4$	$8n-8$ (pentanomial), $4n-4$ (trinomial)
Multiplication in $\mathbb{F}_2[X]$ based on three way split [7]	$\mathbf{M}_2$	$6.27n^{\log_3(6)} - 6.67n + 1.4$
Multiplication one input in $\mathbb{F}_2[X]$ and the other in $\mathbb{F}_4[X]$ [6]	$\mathbf{M}_{2,4}$	$30.75n^{\log_3(5)} - 2.33n\log_3(n) - 34n + 10.5$
Square root in $\mathbb{F}_2[X]$	**sqrt**	$(n-1)/2$

6.2 Applying CANH on Pairing

Bilinear pairing has numorous applications; examples include one round three-party key agreement protocol, identity based encryption, and aggregate signatures. For an excellent introduction to pairing based cryptography the reader can refer to [4]. Many types of pairing exist but the basic types, upon which other types depend, are Weil and Tate pairings. Eta pairing, a variation of Tate pairing, can be written in terms of Tate pairing. For a well-written dictionary of different types of pairing from mathematical point of view, the reader should refer to [49].

In [46], a high speed hardware implementation of Eta pairing is proposed. Toepliz matrix vector product methodology is also given in [46] to increase the speed further. In [6] an idea to utilize field extension polynomial multiplication to speed up elliptic curve algorithms is investigated. They combine multiplications that have a common factor into a multiplication of a higher field to speed up the point doubling and point addition algorithms. As a result, they could have faster point multiplication, which is the core operation of any cryptographic protocol based on elliptic curve cryptography. We observe a common factor pattern within the most expensive operation in Miller's loop, which is the main loop in a pairing algorithm, such as Eta pairing. We develop explicit formulæ for the previous and proposed Miller's loop. We analyze the results and compute the theoretical cost.

In Section 6.2.1 the Eta pairing algorithm is described with emphasis on Miller's loop

and on one special multiplication inside it. In Section 6.2.2, our methodology of improving Miller's loop and consequently Eta pairing is presented. Then, Section 6.2.3 compares the of arithmetic cost of both the previous and the proposed algorithms.

6.2.1 Eta pairing algorithm

Consider a supersingular curve E defined by $Y^2 + Y = X^3 + X$ over the field $\mathbb{F}_{2^{1223}}$. The number of points on the curve, i.e., $\#E(\mathbb{F}_{2^{1223}}) = 5r$ where $r = (2^{1223} + 2^{512} + 1)/5$ [50]. The embedding degree of the curve E is $k = 4$; therefore, the extension field $\mathbb{F}_{2^{4\cdot 1223}}$ is used. This field is constructed using two degree-2 extensions: $\mathbb{F}_{2^{2\cdot 1223}} = \mathbb{F}_{2^{1223}}[\alpha]/(\alpha^2 + \alpha + 1)$ and $\mathbb{F}_{2^{4\cdot 1223}} = \mathbb{F}_{2^{2\cdot 1223}}[\beta]/(\beta^2 + \beta + \alpha)$.

Consider a point $P \in E(\mathbb{F}_{2^{1223}})$ of order r and consider the subgroup μ_r of $\mathbb{F}^*_{2^{4\cdot 1223}}$ which has an order of r. The Eta (η_T) pairing is defined

$$\eta_T : \langle P \rangle \times \langle P \rangle \rightarrow \mu_r$$

as $\eta_T(P_1, P_2) = e(P_1, \psi(P_2))$, such that e is the Tate pairing and $\psi(x, y) = (x + \alpha^2, y + x\alpha + \beta)$. Barreto et al. proposed Algorithm 6.1 for the computation of the η_T pairing [50].

Algorithm 6.1 η_T pairing [50]

Input: $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2) \in E(\mathbb{F}_{2^{1223}})[r]$.
Output: $\eta_T(P_1, P_2)$.

```
T ← x1 + 1
f ← T · (x1 + x2 + 1) + y1 + y2 + (T + x2)α + β
for i = 1 to 612 do
    T ← x1, x1 ← √x1, y1 ← √y1
    g ← T · (x1 + x2) + y1 + y2 + x1 + 1 + (T + x2)α + β
    f ← f · g
    x2 ← x2^2, y2 ← y2^2
end forreturn (f^(2^(2·1223)−1)(2^1223−2^612+1))
```

The computations that will take almost all the time are inside the **for** loop which is a re-expression of the Miller loop of the Tate pairing. We refer to it in this work as the Miller loop for simplicity. In the next section more analysis of this loop is provided.

Miller's loop

Inside the **for** loop, there are two square root computations in the first step. There is one multiplication in $\mathbb{F}_{2^{1223}}$ $(T \cdot (x_1 + x_2 + 1))$ in the second step and it also includes five additions in order to compute g. The third step is a special multiplication $f \cdot g$ in $\mathbb{F}_{2^{4 \cdot 1223}}$ to update f. Finally, there are two squaring computations in the last step. The $f.g$ special multiplication is expressed in terms of additions and multiplications in $\mathbb{F}_{2^{1223}}$. The next section elaborates further.

The $f.g$ multiplication

Due to the special form of $g = g_0 + g_1\alpha + \beta$, the $\mathbb{F}_{2^{4 \cdot 1223}}$ multiplication is reduced to two $\mathbb{F}_{2^{2 \cdot 1223}}$ multiplications plus a number of additions [46]. We write $f = f_0 + f_1\alpha + f_2\beta + f_3\alpha\beta$, then $f \cdot g$ is expressed:

$$\begin{aligned} fg &= (f_0 + f_1\alpha)(g_0 + g_1\alpha) + (f_2 + f_3\alpha)(g_0 + g_1\alpha)\beta + (f_0 + f_1\alpha + f_2\beta + f_3\alpha\beta)\beta \\ &= ((f_0 + f_1\alpha)(g_0 + g_1\alpha) + f_3 + (f_2 + f_3)\alpha) + ((f_2 + f_3\alpha)(g_0 + g_1\alpha) + f_0 + f_2 \\ &\quad + (f_1 + f_3)\alpha)\beta \end{aligned}$$

Basically, there are two multiplications in $\mathbb{F}_{2^{2 \cdot 1223}}$: $(f_0 + f_1\alpha)(g_0 + g_1\alpha)$ and $(f_2 + f_3\alpha)(g_0 + g_1\alpha)$ in addition to adding the results to $f_3 + (f_1 + f_3)\alpha$ and $f_0 + f_2 + (f_1 + f_3)\alpha$. We need to look at the cost of multiplication in $\mathbb{F}_{2^{2 \cdot 1223}}$.

Using the Karatsuba formula, one multiplication is reduced in $\mathbb{F}_{2^{2 \cdot 1223}}$ to three multiplications and four additions in $\mathbb{F}_{2^{1223}}$ by considering the elements of $\mathbb{F}_{2^{2 \cdot 1223}} = \mathbb{F}_{2^{1223}}[\alpha]/(\alpha^2 + \alpha + 1)$ of degree one polynomials. If we have two elements in $\mathbb{F}_{2^{2 \cdot 1223}}$: $U = U_0 + U_1\alpha$ and $V = V_0 + V_1\alpha$, then we write the three products as $P_0 = U_0V_0$, $P_1 = (U_0 + U_1)(V_0 + V_1)$ and $P_2 = U_1V_1$. To get the final result $C = U \times V$, we combine these products in reconstruction step as shown:

$$\begin{aligned} C &= P_0 + (P_1 + P_0 + P_2)\alpha + P_2\alpha^2 \\ &= P_0 + P_2 + (P_1 + P_0)\alpha \mod (\alpha^2 + \alpha + 1) \end{aligned}$$

Therefore, $f \cdot g$ requires 6 multiplications and 15 additions in $\mathbb{F}_{2^{1223}}$.

6.2.2 Improving Miller's loop

In order to derive explicit formulæ of the two $\mathbb{F}_{2^{2\cdot 1223}}$ needed to compute the $f \cdot g$ over $\mathbb{F}_{2^{4\cdot 1223}}$ inside the Miller loop, the Karatsuba scheme is applied [46]:

$$\begin{aligned}(f_0 + f_1\alpha)(g_0 + g_1\alpha) &= f_0g_0 + ((f_0 + f_1)(g_0 + g_1) + f_0g_0 + f_1g_1)\alpha + f_1g_1\alpha^2 \\ (f_2 + f_3\alpha)(g_0 + g_1\alpha) &= f_2g_0 + ((f_2 + f_3)(g_0 + g_1) + f_2g_0 + f_3g_1)\alpha + f_3g_1\alpha^2\end{aligned}$$

The six needed multiplications are

$$\begin{aligned}t_1 &= f_0g_0, & t_2 &= f_2g_0, \\ t_3 &= (f_0 + f_1)(g_0 + g_1), & t_4 &= (f_2 + f_3)(g_0 + g_1), \\ t_5 &= f_1g_1, & t_6 &= f_3g_1.\end{aligned}$$

The following formulæ summarize the Miller loop explicit formulæ after reducing the $\mathbb{F}_{2^{4\cdot 1223}}$ multiplication:

$$\begin{aligned}&T = x_1, \quad x_1 = \sqrt{x_1}, \quad y_1 = \sqrt{y_1}, \\ &g_0 = T \cdot (x_1 + x_2) + y_1 + y_2 + x_1 + 1, \quad g_1 = T + x_2, \\ &t_1 = f_0g_0, \quad t_2 = f_2g_0, \quad g_{01} = g_0 + g_1, \\ &t_3 = (f_0 + f_1)g_{01}, \quad t_4 = (f_2 + f_3)g_{01}, \\ &t_5 = f_1g_1, \quad t_6 = f_3g_1, \\ &w_0 = t_1 + t_5 + f_3, \quad w_1 = t_3 + t_1 + f_2 + f_3, \\ &w_2 = t_2 + t_6 + f_0 + f_2, \quad w_3 = t_4 + t_2 + f_1 + f_3, \\ &f_0 = w_0, \quad f_1 = w_1, \quad f_2 = w_2, \quad f_3 = w_3, \\ &x_2 = x_2^2, \quad y_2 = y_2^2.\end{aligned}$$

Note that since f can be expressed as $f_0 + f_1\alpha + f_2\beta + f_3\alpha\beta$, only f_i where $0 \leq i \leq 3$ are present in the explicit formulæ. Since $g = g_0 + g_1\alpha + \beta$, then $g_2 = 1$ and $g_3 = 0$, and they do not need to be explicitly present in the formulæ. We can easily see that the Miller loop inside the η_T pairing requires $2\,\mathbf{sqrt} + 7\mathbf{M}_2 + 7\mathbf{R}_2 + 2\mathbf{S} + 20\,\mathbf{Add}$.

In the six multiplications to compute t_i values where $1 \leq i \leq 6$, g_0 is common for t_1 and t_2 multiplications, $(g_0 + g_1)$ is common for t_3 and t_4, and t_5 and t_6 have the term g_1 in common. We use a single multiplication in $\mathbb{F}_4$ to replace two multiplications in $\mathbb{F}_2$ [6]. The

field extension based three way split multiplication is described in [7]. As noted earlier, we have $\mathbb{F}_4 = \mathbb{F}_2[\delta]/(\delta^2 + \delta + 1)$. We express these multiplications in the following way:

$$\begin{array}{lcllcllcl} M_1 & = & g_0(f_0 + f_2\delta), & & M_2 & = & M_{1,0}, & M_3 & = & M_{1,1} \\ M_4 & = & (g_0 + g_1)(f_0 + f_1 + (f_2 + f_3)\delta), & & M_5 & = & M_{4,0}, & M_6 & = & M_{4,1} \\ M_7 & = & g_1(f_1 + f_3\delta), & & M_8 & = & M_{7,0}, & M_9 & = & M_{7,1} \end{array}$$

Note that the multiplication results M_i where $i \in \{1, 4, 7\}$ are in $\mathbb{F}_4[X]$ while the other values of the multiplications are in $\mathbb{F}_2[X]$. These values are set to $M_{i,t}$ where $t \in \{0, 1\}$, the 0 in t corresponds to the non-δ part, the 1 in t indexes the δ part. We then write the explicit formulæ after the modification of the Miller loop in η_T pairing as follows:

$$\begin{array}{l} T = x_1, \quad x_1 = \sqrt{x_1}, \quad y_1 = \sqrt{y_1}, \\ g_0 = T \cdot (x_1 + x_2) + y_1 + y_2 + x_1 + 1, \quad g_1 = T + x_2, \\ M_1 = g_0(f_0 + f_2\delta), \\ M_2 = M_{1,0}, \quad M_3 = M_{1,1} \\ M_4 = (g_0 + g_1)(f_0 + f_1 + (f_2 + f_3)\delta), \\ M_5 = M_{4,0}, \quad M_6 = M_{4,1} \\ M_7 = g_1(f_1 + f_3\delta), \\ M_8 = M_{7,0}, \quad M_9 = M_{7,1} \\ w_0 = M_2 + M_8 + f_3, \quad w_1 = M_5 + M_2 + f_2 + f_3, \\ w_2 = M_3 + M_9 + f_0 + f_2, \quad w_3 = M_6 + M_3 + f_1 + f_3, \\ f_0 = w_0, \quad f_1 = w_1, \quad f_2 = w_2, \quad f_3 = w_3, \\ x_2 = x_2^2, \quad y_2 = y_2^2. \end{array}$$

As in the above formulæ, the six multiplications in $\mathbb{F}_2[X]$ are replaced by three multiplications in $\mathbb{F}_4[X]$. We can notice that the total cost of Miller's loop after the modification is $2\,\mathbf{sqrt} + 1\mathbf{M}_2 + 1\mathbf{R}_2 + 3\mathbf{M}_{2,4} + 3\mathbf{R}_4 + 2\mathbf{S} + 20\,\mathbf{Add}$.

6.2.3 Cost comparison

We consider the cost of the square root to be equal to $(n-1)/2$. In [46], a methodology is suggested to perform square root in $\mathbb{F}_{2^{1223}}$, so if we have an element $A = \sum_{i=0}^{1223} a_i x^i$ in

$\mathbb{F}_{2^{1223}}$, then we can write

$$\sqrt{A} = \sqrt{\sum_{i=0}^{611} a_{2i}x^{2i}} + \sqrt{\sum_{i=0}^{610} a_{2i+1}x^{2i+1}} = \left(\sum_{i=0}^{611} a_{2i}x^{i}\right) + \sqrt{x}\left(\sum_{i=0}^{610} a_{2i+1}x^{i}\right).$$

However, $x = x^{256}+x^{1224} \mod (1+x^{255}+x^{1223})$, so $\sqrt{x} = x^{128}+x^{612} \mod (1+x^{255}+x^{1223})$. Hence, $\sqrt{A} = \sum_{i=0}^{127} a_{2i}x^i + \sum_{i=128}^{611}(a_{2i}+a_{2i-256+1})x^i + \sum_{i=0}^{126}(a_{2i+1}+a_{2(i+612-128)+1})x^{i+612} + \sum_{i=127}^{610} a_{2i+1}x^{i+612}$. The number of XOR gates needed (or additions) is 611 which is $\frac{n-1}{2}$ knowing that $n = 1223$.

The total arithmetic cost of a single round of Miller's loop using the previous technique is $43.89n^{\log_3(6)} - 3.69n - 41.2$. After utilizing the field extension multiplication, the total arithmetic cost is $6.27n^{\log_3(6)} + 92.25n^{\log_3(5)} - 6.99n\log(n) - 65.67n - 18.1$. Table 6.2 summarizes the cost of Miller's loop in terms of both basic operations and arithmetic cost.

Table 6.2: The cost of one round of the Miller loop in η_T pairing using the previous and the proposed techniques

Miller loop method	**No. of basic operations**	**Arithmetic cost**
previous	$2\,\mathbf{sqrt} + 7\mathbf{M}_2 + 7\mathbf{R}_2 + 2\mathbf{S} + 20\,\mathbf{Add}$	$43.89n^{\log_3(6)} - 3.69n - 41.2$
proposed	$2\,\mathbf{sqrt} + 1\mathbf{M}_2 + 1\mathbf{R}_2 + 3\mathbf{M}_{2,4} + 3\mathbf{R}_4 + 2\mathbf{S} + 20\,\mathbf{Add}$	$6.27n^{\log_3(6)} + 92.25n^{\log_3(5)} - 6.99n\log_3(n) - 65.67n - 18.1$

By substituting $n = 1223$ as in the algorithm, the total arithmetic cost for the previous formulæ is 4756918 and that of the proposed formulæ is 3620404. Therefore, we observe an improvement of **23.9%** by applying the common factor technique. Almost all the cost of the η_T pairing algorithm is a direct result of Miller's loop. Improving Miller's loop in Algorithm 6.1 will likely to significantly increase the speed of this algorithm because it has 611 iterations.

6.2.4 Software implementation and results

In order to verify the theoretical results, we implemented the pairing algorithm in software. We used C♯ as programing language with visual studio as development framework.

The personal laptop used is Intel(R) core(TM) i7-3632QM and has speed of 2.2 GH. The operating system is Windows 8 home edition. Algorithm 6.1 has been implemented which contains both the Miller loop and the final exponentiation. Refer to [46] for details of the algorithms involved under the final exponentiation. The CANH based version of the pairing has also been implemented.

The experiment ran 50 times for each of the different pairing algorithms enabling us to compute the average. The typical pairing algorithm takes 344835765 clock cycles. After applying the CANH trick, the algorithm takes 250789374 clock cycles. Therefore, applying CANH trick into pairing results in an improvement of **27.27%**.

6.3 Applying CANH on Hyperelliptic Curves

The CANH trick works over binary fields and uses a 3-way split polynomial multiplication algorithm [6]. Similar to elliptic curves, hyperelliptic curves have group operations, which are addition and doubling. However, here the operations are applied to the divisor over the Jacobian of hyperelliptic curves. In [47], the fastest explicit formulæ for hyperelliptic curves over even characteristic genus 2 hyperelliptic curves using projective coordinates are presented. In this paper, we apply the CANH trick over the explicit formulæ given in [47].

Cantor, in his breakthrough paper in 1987, introduced an algorithm to make computations in the Jacobian of hyperelliptic curve for doing group laws operations such as addition [51]. His work focuses on odd characteristics, but Koblitz in his paper in 1988 extended his work to even characteristics and introduced a complete hyperelliptic cryptosystem [52]. A number of references provide excellent background about hyperelliptic curves; for example see [5] and [53].

Harley [54] has significantly optimized Cantor's algorithm. In [55], there are efficient explicit formulæ for group operations for hyperelliptic curves over genus 2. In [56], Lange introduces new explicit formulæ that are inversion free by using projective coordinates. Lange's work in [57] improves Harley's work and also provides explicit formulæ for computations over genus 2. The work of Wollingor and Kovtun [47] improves the explicit formulæ of hyperelliptic curves of both even and odd characteristics.

The work of Costello and Lauter [58] has better explicit formulæ than [47] but only in odd characteristics. The work of Avanzi et al. improves the explicit formulæ in genus 3

and genus 4. The work of Gaudry [59] proposes a very fast algorithm for genus 2 arithmetic but in the Kummer surface using theta functions.

In this section we will show the use the CANH trick [6] in hyperelliptic curves. It is based on using field extension 3-way split polynomial multiplication algorithm to replace two multiplications in $\mathbb{F}_2$ by one in $\mathbb{F}_4$. The complexity of the field extension multiplication is less than two multiplications. The field extension multiplication is explained in [7]. This work considers only binary fields so it is now applied to even characteristic hyperelliptic curves. The work of Wollinger and Kovtun [47] provides the fastest explicit formulæ on even characteristic over genus two hyperelliptic curves; therefore, we apply the CANH trick on their formulæ.

Section 6.3.1 shows the previous and proposed addition formulæ of divisors. The formulæ of doubling before and after applying CANH trick are shown in Section 6.3.2. In Section 6.3.3, complexity comparison and the improvement of using this trick are highlighted.

6.3.1 Addition

The following are the explicit formulæ given in [47] for addition over mixed coordinates for even characteristics. In the mixed coordinates system, one input is in projective coordinates and the other is in affine coordinates; that is, it has the Z value equal to one. Using mixed coordinate saves more operations than projective coordinates. In affine coordinates, an inversion is needed which is usually a costly operation. We observe that this set of addition formulæ has a cost of 37 $\mathbf{M}_2$, 37 $\mathbf{R}_2$, 5 $\mathbf{S}$, and 27 **Add**.

$$
\begin{aligned}
&\widetilde{U}_{11} = Z_2 \cdot U_{11}, \quad y_1 = \widetilde{U}_{11} + U_{21}, \quad y_2 = U_{20} + U_{10} \cdot Z_2, \quad y_3 = y_1 \cdot U_{11} + y_2,\\
&r = y_2 \cdot y_3 + y_1^2 \cdot U_{10}, \quad \mathrm{inv}_1 = y_1, \quad \mathrm{inv}_0 = y_3, \quad w_0 = V_{10} \cdot Z_2 + V_{20},\\
&w_1 = V_{11} \cdot Z_2 + V_{21}, \quad w_2 = \mathrm{inv}_0 \cdot w_0, \quad w_3 = \mathrm{inv}_1 \cdot w_1, \quad s_0 = w_2 + U_{10} \cdot w_3,\\
&s_1 = (\mathrm{inv}_0 + \mathrm{inv}_1) \cdot (w_0 + w_1) + w_2 + w_3 \cdot U_{11}, \quad R = r \cdot Z_2, \quad s_2 = s_0 \cdot Z_2,\\
&s_3 = s_1 \cdot Z_2, \quad \widetilde{R} = R \cdot s_3, \quad w_0 = s_1 \cdot s_0, \quad w_1 = s_1 \cdot s_3 \quad w_2 = s_0 \cdot s_3, \quad w_3 = w_1 \cdot U_{21},\\
&w_4 = R \cdot s_1, \quad l_0 = w_0 \cdot U_{20}, \quad l_2 = w_3 + w_2, \quad l_1 = (w_1 + w_0) \cdot (U_{21} + U_{20}) + l_0 + w_3,\\
&\widetilde{U}'_0 = s_2^2 + s_1^2 \cdot y_1 U_{11} + y_2 \cdot w_1 + \widetilde{R} + R \cdot r \cdot y_1, \quad \widetilde{U}'_1 = w_1 \cdot y_1 + R^2, \quad U'_0 = \widetilde{U}'_0 \cdot \widetilde{R},\\
&U'_1 = \widetilde{U}'_1 \cdot \widetilde{R}, \quad Z' = s_3^2 \cdot \widetilde{R}, \quad V'_1 = \widetilde{U}'_1 \cdot (l_2 + \widetilde{U}'_1) + s_3^2 \cdot (\widetilde{U}'_0 + w_4 \cdot V_{21} + l_1),\\
&V'_0 = \widetilde{U}'_0 \cdot (l_2 + \widetilde{U}'_1) + s_3^2 \cdot (l_0 + w_4 \cdot V_{20}).
\end{aligned}
$$

Table 6.3: Summary of common factors in divisor addition.

Multiplications	Common Factor	Multiplications	Common Factor
$Z_2 \cdot U_{11}, \quad V_{10} \cdot Z_2$	Z_2	$U_{10} \cdot Z_2, \quad V_{11} \cdot Z_2$	Z_2
$y_1 \cdot U_{11}, \quad \mathrm{inv}_1 \cdot w_1$	y_1 [a]	$y_2 \cdot y_3, \quad \mathrm{inv}_0 \cdot w_0$	y_3 [b]
$y_1^2 \cdot U_{10}, \quad U_{10} \cdot w_3,$	U_{10}	$r \cdot Z_2, \quad r \cdot y_1$	r
$s_0 \cdot Z_2, \quad s_1 \cdot Z_2$	Z_2	$s_1 \cdot s_0, \quad s_0 \cdot s_3$	s_0
$R \cdot s_3, \quad R \cdot r y_1$	R	$w_1 \cdot U_{21}, \quad y_2 \cdot w_1$	w_1
$R \cdot r \cdot y_1, \quad w_1 y_1$	y_1	$\widetilde{U}'_0 \cdot \widetilde{R}, \quad \widetilde{U}'_1 \cdot \widetilde{R}$	$\widetilde{R}$
$\widetilde{U}'_1 \cdot (l_2 + \widetilde{U}'_1), \quad \widetilde{U}'_0 \cdot (l_2 + \widetilde{U}'_1)$	$l_2 + \widetilde{U}'_1$	$w_4 \cdot V_{21}, \quad w_4 \cdot V_{20}$	w_4
$s_3^2 \cdot (\widetilde{U}'_0 + w_4 \cdot V_{21} + l_1), \quad s_3^2 \cdot (l_0 + w_4 \cdot V_{20})$	s_3^2		

[a] since $\mathrm{inv}_1 = y_1$
[b] since $\mathrm{inv}_0 = y_3$

We find 15 pairs of multiplications which have common factors between them. Table 6.3 summarizes the common factors in the addition formulæ. Note that we utilize inv_1 and inv_0 as common factors with y_1 and y_3 respectively, because they hold the same values and they are not changed before the use of common factor.

The following formulæ are the resultant divisor addition formulæ after applying the CANH trick. Using the 15 common factors, 30 multiplications in $\mathbb{F}_2$ are replaced by 15 multiplications where one input is in $\mathbb{F}_2$ and the other is in $\mathbb{F}_4$ ($M_{2,4}$). The output of this multiplication is in $\mathbb{F}_4$; that is why a reduction in $\mathbb{F}_4$ is needed. In summary, the total cost is 15 $\mathbf{M}_{2,4}$, 15 $\mathbf{R}_4$, 7 $\mathbf{M}_2$, 7 $\mathbf{R}_2$, 15 $\mathbf{S}$, and 27 **Add**.

$M_1 = Z_2(U_{11} + V_{10}\delta)$, $\widetilde{U}_{11} = M_{1,0}$, $y_1 = \widetilde{U}_{11} + U_{21}$, $w_0 = M_{1,1} + V_{20}$,
$\mathrm{inv}_1 = y_1$, $M_2 = Z_2(U_{10} + V_{11}\delta)$, $y_2 = U_{20} + M_{2,0}$, $w_1 = M_{2,1} + V_{21}$,
$M_3 = y_1(U_{11} + w_1\delta)$, $y_3 = M_{3,0} + y_2$, $\mathrm{inv}_0 = y_3$, $w_3 = M_{3,1}$,
$M_4 = y_3(y_2 + w_0\delta)$, $w_2 = M_{4,1}$, $M_5 = U_{10}(y_1^2 + w_3\delta)$, $r = M_{4,0} + M_{5,0}$,
$s_0 = w_2 + M_{5,0}$, $s_1 = (\mathrm{inv}_0 + \mathrm{inv}_1) \cdot (w_0 + w_1) + w_2 + w_3 \cdot U_{11}$, $M_6 = r(Z_2 + y_1\delta)$,
$R = M_{6,0}$, $M_7 = Z_2(s_0 + s_1\delta)$, $s_2 = M_{7,0}$, $s_3 = M_{7,1}$, $M_8 = s_0(s_1 + s_3\delta)$,
$w_0 = M_{8,0}$, $w_2 = M_{8,1}$, $M_9 = s_3(R + s_1\delta)$, $\widetilde{R} = M_{9,0}$, $w_1 = M_{9,1}$,
$M_{10} = w_1(U_{21} + y_2\delta)$, $w_3 = M_{10,0}$, $l_0 = w_0 \cdot U_{20}$, $l_2 = w_3 + w_2$,
$l_1 = (w_1 + w_0) \cdot (U_{21} + U_{20}) + l_0 + w_3$, $M_{11} = R(s_1 + M_{6,1}\delta)$,
$\widetilde{U}_0' = s_2^2 + s_1^2 \cdot y_1 U_{11} + M_{10,1} + \widetilde{R} + M_{11,1}$, $\widetilde{U}_1' = w_1 \cdot y_1 + R^2$, $M_{12} = \widetilde{R}(\widetilde{U}_0' + \widetilde{U}_1'\delta)$,
$U_0' = M_{12,0}$, $U_1' = M_{12,1}$, $Z' = s_3^2 \cdot \widetilde{R}$, $M_{13} = (l_2 + \widetilde{U}_1')(\widetilde{U}_1' + \widetilde{U}_0'\delta)$,
$M_{14} = w_4(V_{21} + V_{20}\delta)$, $M_{15} = s_3^2((\widetilde{U}_0' + M_{14,0} + l_1) + (l_0 + M_{14,1})\delta)$,
$V_1' = M_{13,0} + M_{14,0}$, $V_0' = M_{13,1} + M_{14,1}$,

6.3.2 Doubling

The following set of formulæ shows the the doubling formulæ as presented in [47]. There are 29 $\mathbf{M}_2$, 29 $\mathbf{R}_2$, 7 $\mathbf{S}$, and 20 **Add**.

$Z_2 = Z^2$, $w_0 = V_1^2$, $w_1 = U_1^2$, $, w_2 = Z \cdot U_1$, $R = U_0 \cdot Z_2^2$, $\mathrm{inv}_1 = Z$,
$\mathrm{inv}_0 = w_2$, $k_1 = w_1$, $k_0 = U_1 \cdot w_1 + Z \cdot (Z \cdot V_1 + w_0)$, $w_0 = k_0 \cdot \mathrm{inv}_0$,
$w_1 = k_1 \cdot Z$, $s_0 = w_0 + Z \cdot U_0 \cdot w_1$, $s_3 = (\mathrm{inv}_0 + Z)(k_0 + k_1) + w_0 + w_1 \cdot (1 + U_1)$,
$\widetilde{R} = R \cdot s_1$, $w_0 = s_1 \cdot s_3$, $w_1 = s_0 \cdot s_3$, $w_3 = w_1 \cdot Z$, $w_4 = R \cdot s_3$, $l_0 = U_0 \cdot w_1$,
$l_2 = U_1 \cdot w_0$, $l_1 = (w_1 + w_2) \cdot (U_1 + U_0) + l_0 + l_2$, $\widetilde{U}_0' = s_0^2 + \widetilde{R}$, $\widetilde{U}_1' = R^2$,
$U_0' = \widetilde{U}_0' \cdot \widetilde{R}$, $U_1' = \widetilde{U}_1' \cdot \widetilde{R}$, $Z' = s_1^2 \cdot \widetilde{R}$,
$V_0' = \widetilde{U}_0' \cdot (l_2 + \widetilde{U}_1' + w_3) + s_1^2 \cdot (l_0 + w_4 \cdot V_0)$,
$V_1' = \widetilde{U}_1' \cdot (l_2 + \widetilde{U}_1' + w_3) + s_1^2 \cdot (\widetilde{U}_0' + \widetilde{R} + w_4 \cdot V_1 + l_1)$,

Table 6.4 shows the common factors of the doubling formulæ such that each pair is combined in a single multiplication of a higher field. We observe that 11 pairs of multiplications have common factors.

Table 6.4: Summary of common factors in divisor doubling.

Multiplications	Common Factor	Multiplications	Common Factor
$Z \cdot U_1, \quad U_1 \cdot w_1$	U_1	$k_1 \cdot Z, \quad Z \cdot V_1$	Z
$Z \cdot w_1, \quad Z \cdot (Z \cdot V_1 + w_0)$	Z	$Z \cdot U_0 \cdot w_1, \quad U_0 \cdot Z_2^2,$	U_0
$s_3 \cdot Z, \quad s_0 \cdot s_3$	s_3	$R \cdot s_1, \quad s_1 \cdot s_3$	s_1
$w_1 \cdot Z, \quad U_0 \cdot w_1$	w_1	$\widetilde{U}_0' \cdot R, \quad \widetilde{U}_1' \cdot R$	R
$\widetilde{U}_0'(l_2 + \widetilde{U}_1 + w_3), \quad \widetilde{U}_1 \cdot (l_2 + \widetilde{U}_1' + w_3)$	$(l_2 + \widetilde{U}_1' + w_3)$	$s_1^2(l_0 + w_4 \cdot V_0), \quad s_1^2 \cdot (V_0' + \widetilde{R} + w_4 \cdot V_1 + l_1)$	s_1^2
$w_4 \cdot V_0, \quad w_4 \cdot V_1$	w_4		

The following set of formulæ shows the proposed doubling formulæ after applying the CANH trick. This formulæ set costs 11 $\mathbf{M}_{2,4}$, 11 $\mathbf{R}_4$, 7 $\mathbf{M}_2$, 7 $\mathbf{R}_2$, 7 $\mathbf{S}$, and 20 **Add**.

$Z_2 = Z^2, \quad w_0 = V_1^2, \quad w_1 = U_1^2, \quad M_1 = U_1(Z + w_1\delta), \quad w_2 = M_{1,0}, \quad \text{inv}_1 = Z,$
$inv_0 = w_2, \quad k_1 = w_1, \quad M_2 = Z(k_1 + V_1\delta), \quad w_1 = M_{2,0},$
$M_3 = Z(w_1 + (M_{2,1} + w_0)\delta), \quad k_0 = M_{1,1} + M_{3,1}, \quad w_0 = k_0 \cdot \text{inv}_0$
$M_4 = U_0(M_{3,0} + Z_2^2\delta), \quad R = M_{4,1}, \quad s_3 = (\text{inv}_0 + Z) \cdot (k_0 + k_1) + w_0 + w_1 \cdot (1 + U_1)$
$s_0 = w_0 + M_{4,1}, \quad M_5 = s_3(Z + s_0\delta), \quad s_1 = M_{5,0}, \quad w_1 = M_{5,1},$
$M_6 = s_1(R + s_3\delta), \quad \widetilde{R} = M_{6,0}, \quad w_0 = M_{6,1}, \quad M_7 = w_1(Z + U_0\delta), \quad w_3 = M_{7,1},$
$w_4 = R \cdot s_3, \quad l_0 = M_{7,1}, \quad l_2 = U_1 \cdot w_0, \quad l_1 = (w_1 + w_2) \cdot (U_1 + U_0) + l_0 + l_2,$
$\widetilde{U}_0' = s_0^2 + \widetilde{R}, \quad \widetilde{U}_1' = R^2, \quad M_8 = \widetilde{R}(\widetilde{U}_0' + \widetilde{U}_1'\delta), \quad U_0' = M_{8,0}, \quad U_1' = M_{8,1},$
$Z' = s_1^2 \cdot \widetilde{R}, \quad M_9 = (l_2 + \widetilde{U}_1' + w_3)(\widetilde{U}_0' + \widetilde{U}_1'\delta), \quad M_{10} = w_4(V_0 + V_1\delta),$
$M_{11} = s_1^2((l_0 + M_{10,0}) + (\widetilde{U}_0' + \widetilde{R} + M_{10,1} + l_1)\delta), \quad V_0' = M_{9,0} + M_{11,0},$
$V_1' = M_{9,1} + M_{11,1}.$

6.3.3 Complexity comparison

Table 6.5 summarizes the total cost of both previous and proposed formulæ for addition and doubling. The total arithmetic complexities of addition and doubling before and after applying the CANH trick are shown in Table 6.6.

Table 6.5: Cost comparison of previous and proposed formulæ.

Formulæ	**Previous**	**Proposed**
Addition	$37\mathbf{M}_2 + 37\mathbf{R}_2 + 5\mathbf{S} + 27\,\mathbf{Add}$	$15\mathbf{M}_{2,4} + 15\mathbf{R}_4 + 7\mathbf{M}_2 + 7\mathbf{R}_2 + 5\mathbf{S} + 27\,\mathbf{Add}$
Doubling	$29\mathbf{M}_2 + 29\mathbf{R}_2 + 7\mathbf{S} + 20\,\mathbf{Add}$	$11\mathbf{M}_{2,4} + 11\mathbf{R}_4 + 7\mathbf{M}_2 + 7\mathbf{R}_2 + 7\mathbf{S} + 20\,\mathbf{Add}$

Table 6.6: Complexity cost comparison of previous and proposed formulæ.

Formulæ	**Previous**	**Proposed**
Addition	$231.99n^{\log_3(6)} - 51.79n - 116.2$	$43.89n^{\log_3(6)} + 461.25n^{\log_3(5)} - 34.95n\log_3(n) - 360.69n + 0.7$
Doubling	$181.83n^{\log_3(6)} - 29.43n - 103.4$	$43.89n^{\log_3(6)} + 338.25n^{\log_3(5)} - 25.63n\log_3(n) - 256.69n - 19.2$

In order to get an idea of how much the new algorithm improves the previous one, figures showing the cost complexity versus the field size (n) are presented. Figure 6.1 shows the addition complexity comparison, and Figure 6.2 shows the complexities in the case of doubling. In both cases, we see that there is an improvement that starts at around 120 bits. Below 108 bits, the proposed scheme will result in a degradation of the overall performance. However, the improvement can go up to 80% at a very large field size.

Figure 6.3, which shows the amount of improvement up to field size of 570 bits, indicates how much improvement can be achieved for this range. In the case of addition, the improvement is a little bit more than that of doubling.

In order to relate Figure 6.3 to the equations in Table 6.6, it is observed that both the previous and the proposed methodologies belong to the same asymptotic complexities, i.e. $O(n^{\log_3(6)})$. The proposed methodology has a lower coefficient with respect to the term $n^{\log_3(6)}$, but it has an additional term $n^{\log_3(5)}$ with a comparatively larger coefficient. The dominant term is $n^{\log_3(6)}$ and, therefore, the limit of improvement is 81% in the case of addition, and 76% in the case of doubling. However, until this limit is reached, the improvement will initially have the burden of the high coefficient of $n^{\log_3(5)}$ term.

Figure 6.1: Addition complexity comparison.

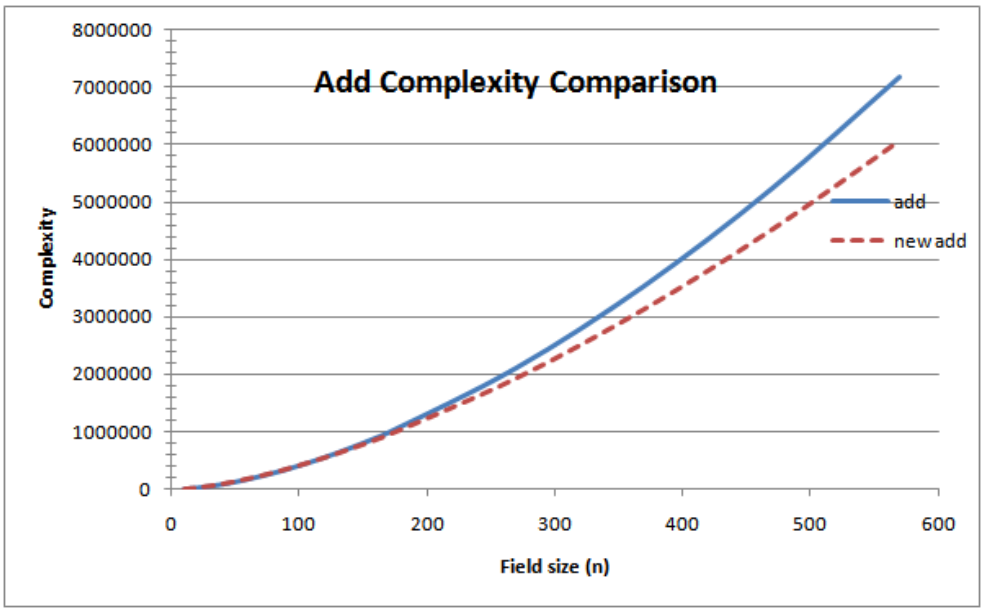


Figure 6.2: doubling complexity comparison.

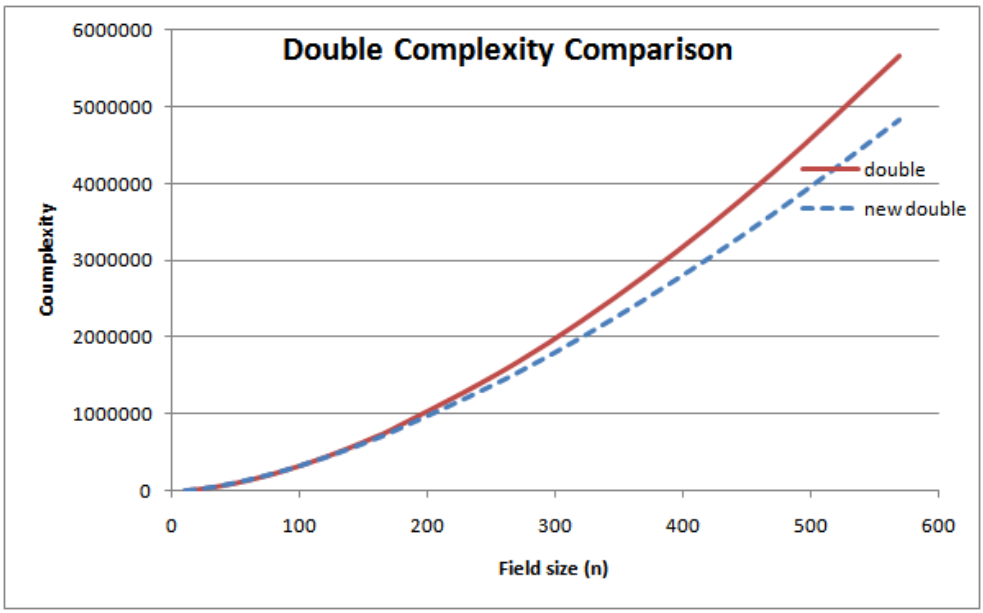


Figure 6.3: Improvement percentage of double and add operations.

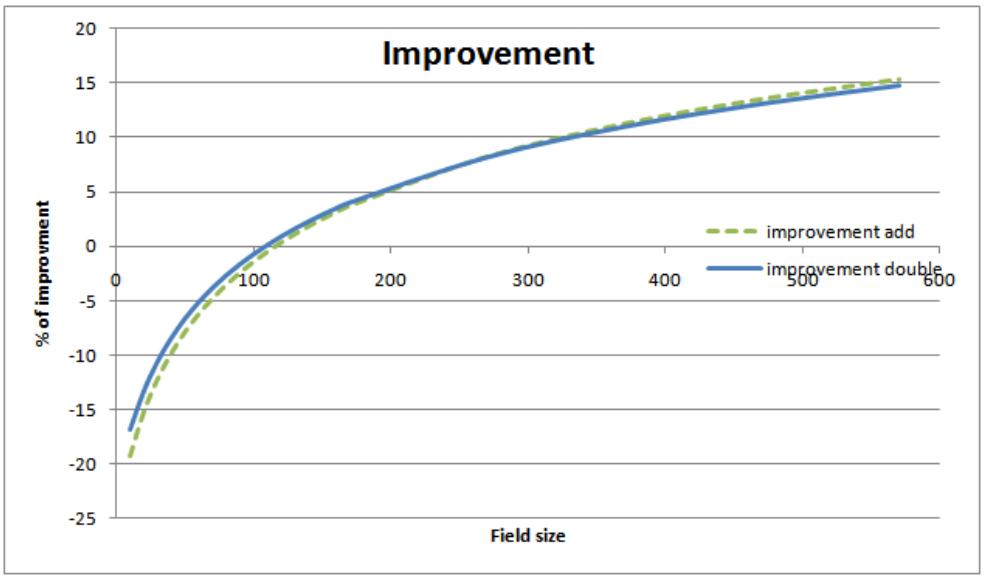


Table 6.7: The improvement using CANH technique in hyperelliptic curves in certain field sizes.

field	Addition improvement	Doubling improvement
163	3.391	3.65
233	6.87	6.86
283	8.75	8.61
409	12.28	11.88
571	15.41	14.79
1032	20.78	19.78

The improvement for specific field sizes are summarized in Table 6.7. These field sizes are the NIST recommended fields in addition to 1032 bits value. The improvement goes from around 3% at 163 to around 20% when the the field size is 1032.

Hyperelliptic curves usually work with a comparatively small key size. In this case, the improvement is not as significant as it is if we go for higher sizes. For example, if the field size is 100-bits, then there would be a degradation by around 1.3% using this technique.

In the case of working in hyperelliptic pairing, the extension field size is required to be high, for example, for 80 bits security level, the extension field size is required to be 1024; while for 256 bit security level, the extension field size is 15360 [60]. For these sizes of curves, the group law operations using the CANH technique achieve very good improvement.

6.4 Summary

In this chapter, we have discussed the use of field extension based 3-way split polynomial multiplication to improve the speed of η_T pairing. Our theoretical analysis shows that an improvement of 23.9% can be achieved using this method. Miller's loop consumes almost all the time in η_T pairing computation. Inside this loop, a multiplication in $\mathbb{F}_{2^{4\cdot 1223}}$ is present. After reducing this multiplication we notice the presence of three pairs of a common factor multiplication. We use a multiplication in $\mathbb{F}_4[X]$ to replace two in $\mathbb{F}_2[X]$. Explicit formulæ

and cost analysis are presented in this chapter.

In addition, in this chapter we have shown how to apply the CANH trick over hyperelliptic curves. The results show good improvement over most of the field sizes. The higher the size the more the improvement to certain threshold.

Chapter 7

Summary and Future Work

7.1 Summary

Polynomial multiplication is crucial in cryptography, especially for those systems that are based on elliptic and hyper-elliptic curves. Several polynomial multiplication algorithms have been studied in this proposal, namely, the K2W, K3W, IK3W, B3W and FE3W algorithms. We have also showed two types of point addition and point doubling algorithms: the conventional and the CANH ones. The CANH algorithms depend on the use of the field extension polynomial multiplication algorithm and it replaces two polynomial multiplications in a lower field when there is a common operand between them. Utilizing CANH trick in pairing and hyperelliptic curves proves to be useful. We found an improvement of 23.9% when using this method with η_T pairing computation. In the case of hyperelliptic curves the use of CANH trick improves after certain threshold of field size, and perform better for larger field size.

7.2 Future Work

In this section, we discuss three ideas for future work. These are CANH improvement and extension, different uses of the common factor trick, and software and hardware realization.

CANH improvement and extension

CANH improves over the 3-way split algorithm, but it might not improve in other split based algorithms. Therefore, the CANH trick shall only be used when it improves over the split algorithm when it is needed in the polynomial multiplication. Making use of the field extension has previously been considered only in 3-way split algorithms. As a result, one direction of the future research is to look into other split based algorithms and see where they can be improved.

Different use of common factor trick

In hyperelliptic curve addition, there is an occurrence of more than two terms that have common factors. For example, if we have four multiplications like AB, AC, AD, and AE, where A is a common factor. It will be interesting to investigate whether n multiplications with one common input can be efficiently performed as a single multiplication where one input is in $\mathbb{F}_2$ and the other in $\mathbb{F}_{2^n}$.

Software/Hardware realization

Several libraries support different cryptography applications. Polymul [61] and zn_poly [62], for example, support efficient multiplication algorithms. Other libraries support number theoretic and integer arithmetic applications like FLINT [63], GMP [64], NTL [65], GNU PG [66]. Libraries support higher level algorithms for certain applications or protocols like elliptic curve cryptography or other communication features like crypto++ [67], Cryptlib [68], Libgcrypt [69], MIRACL [70], LiDIA [71], OpenSSL [72], and PARI-GP [73]. In [74], one can find an excellent comparison between different software libraries used for public key cryptography. It would be interesting to implement and optimize in software and hardware the newer algorithms studied in this thesis and make a thorough comparison.

APPENDICES

Appendix A

Reduction Algorithms

Algorithm A.1 Reduction modulo $f(z) = z^{233} + z^{74} + 1$.

1: **procedure** REDUCE233(x) ▷ x is a polynomial of size maximum $2m - 2$ where $m = 233$.
2: $ones \leftarrow 2^{233} - 1$.
3: $temp \leftarrow x \gg 233$.
4: $result \leftarrow x \,\&\, ones$
5: $result \leftarrow result \oplus (temp \ll 74) \oplus temp$.
6: $temp \leftarrow result \gg 233$.
7: $result \leftarrow result \oplus (temp \ll 74) \oplus temp$.
8: $result \leftarrow result \,\&\, ones$.
9: **return** $result$.
10: **end procedure**

Algorithm A.2 Reduction modulo $f(z) = z^{283} + z^{12} + z^5 + 1$.

1: **procedure** REDUCE283(x) ▷ x is a polynomial of size maximum $2m - 2$ where $m = 283$.
2: $ones \leftarrow 2^{283} - 1$.
3: $temp \leftarrow x \gg 283$.
4: $result \leftarrow x$ & $ones$
5: $result \leftarrow result \oplus (temp \ll 12) \oplus (temp \ll 7) \oplus (temp \ll 5) \oplus temp$.
6: $temp \leftarrow result \gg 283$.
7: $result \leftarrow result \oplus (temp \ll 12) \oplus (temp \ll 7) \oplus (temp \ll 5) \oplus temp$.
8: $result \leftarrow result$ & $ones$.
9: **return** $result$.
10: **end procedure**

Algorithm A.3 Reduction modulo $f(z) = z^{409} + z^{87} + 1$.

1: **procedure** REDUCE409(x) ▷ x is a polynomial of size maximum $2m - 2$ where $m = 409$.
2: $ones \leftarrow 2^{409} - 1$.
3: $temp \leftarrow x \gg 409$.
4: $result \leftarrow x$ & $ones$
5: $result \leftarrow result \oplus (temp \ll 87) \oplus temp$.
6: $temp \leftarrow result \gg 409$.
7: $result \leftarrow result \oplus (temp \ll 87) \oplus temp$.
8: $result \leftarrow result$ & $ones$.
9: **return** $result$.
10: **end procedure**

Algorithm A.4 Reduction modulo $f(z) = z^{512} + z^{10} + z^5 + z^2 + 1$.

1: **procedure** REDUCE512(x) ▷ x is a polynomial of size maximum $2m - 2$ where $m = 512$.
2: $ones \leftarrow 2^{512} - 1$.
3: $temp \leftarrow x \gg 512$.
4: $result \leftarrow x \ \& \ ones$
5: $result \leftarrow result \oplus (temp \ll 12) \oplus (temp \ll 7) \oplus (temp \ll 5) \oplus temp$.
6: $temp \leftarrow result \gg 512$.
7: $result \leftarrow result \oplus (temp \ll 12) \oplus (temp \ll 7) \oplus (temp \ll 5) \oplus temp$.
8: $result \leftarrow result \ \& \ ones$.
9: **return** $result$.
10: **end procedure**

References

[1] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1055638

[2] R. R. L., A. Shamir, and L. Adlman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[3] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *Information Theory, IEEE Transactions on*, vol. 31, no. 4, pp. 469 – 472, jul 1985.

[4] A. Menezes, "An introduction to pairing-based cryptography," *Recent Trends in Cryptography, Contemporary Mathematics*, vol. 477, pp. 47–65, 2009.

[5] A. J. Menezes, Y.-H. Wu, and R. J. Zuccherato, "An elementary introduction to hyperelliptic curves," University of Waterloo, Tech. Rep. CORR 96-19, 1996.

[6] M. Cenk, A. S. Alrefai, C. Negre, and M. A. Hasan, "A new approach to low complexity point addition/multiplication on binary elliptic curves," 2012.

[7] M. Cenk, C. Negre, and M. Hasan, "Improved three-way split formulas for binary polynomial and toeplitz matrix vector products," 2012.

[8] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. New York: Springer, 2004.

[9] "Federal information processing standards publication 186-3," National Institute of Standards and Technology, DIGITAL SIGNATURE STANDARD, 2009. [Online]. Available: http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

[10] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209.

[11] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology CRYPTO 85 Proceedings*, ser. Lecture Notes in Computer Science, H. Williams, Ed. Springer Berlin / Heidelberg, 1986, vol. 218, pp. 417–426. [Online]. Available: http://dx.doi.org/10.1007/3-540-39799-X_31

[12] D. J. Bernstein and T. Lange, "Explicit-formula database," 6 2012, http://www.hyperelliptic.org/EFD/.

[13] M. Cenk, C. K. Koc, and F. Ozbudak, "Polynomial multiplication over finite fields using field extensions and interpolation," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on.* IEEE, 2009, pp. 84–91.

[14] A. Weimerskirch and C. Paar, "Generalizations of the karatsuba algorithm for efficient implementations," Cryptology ePrint Archive, Tech. Rep. 2006/224, 2006.

[15] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," in *Soviet Mathematics Doklady*, vol. 3, no. 4, 1963, pp. 714–716.

[16] S. A. Cook and S. O. Aanderaa, "On the minimum computation time of functions," *Transactions of the American Mathematical Society*, vol. 142, pp. 291–314, 1969.

[17] S. Winograd, *Arithmetic Complexity of Computations.* Society For Industrial & Applied Mathematics, U.S., 1980.

[18] D. Bernstein, "Batch binary edwards," in *Advances in Cryptology - CRYPTO 2009*, ser. Lecture Notes in Computer Science, S. Halevi, Ed. Springer Berlin / Heidelberg, 2009, vol. 5677, pp. 317–336.

[19] N. Smart, "A comparison of different finite fields for elliptic curve cryptosystems," *Computers & Mathematics with Applications*, vol. 42, no. 1, pp. 91–100, 2001.

[20] J. López and R. Dahab, "High-speed software multiplication in f2m," in *Progress in CryptologyINDOCRYPT 2000*. Springer, 2000, pp. 203–212.

[21] D. J. Bernstein, "Multidigit multiplication for mathematicians," *URL: http://cr. yp. to/papers. html*, 2001.

[22] H. Fan, J. Sun, M. Gu, and K.-Y. Lam, "Overlap-free karatsuba-ofman polynomial multiplication algorithms," *Information Security, IET*, vol. 4, no. 1, pp. 8–14, 2010.

[23] H. Fan and M. A. Hasan, "A new approach to subquadratic space complexity parallel multipliers for extended binary fields," *Computers, IEEE Transactions on*, vol. 56, no. 2, pp. 224–233, 2007.

[24] C. Negre, "Efficient binary polynomial multiplication based on optimized karatsuba reconstruction," 2012.

[25] ——, "Improved three-way split approach for binary polynomial multiplication based on optimized reconstruction," 2013.

[26] R. Avanzi and N. Thériault, "Effects of optimizations for software implementations of small binary field arithmetic," in *Arithmetic of Finite Fields*. Springer, 2007, pp. 69–84.

[27] D. F. Aranha, J. Lopez, and D. Hankerson, "Efficient software implementation of binary field arithmetic using vector instruction sets," *Progress in Cryptology - Latincrypt 2010*, vol. 6212, pp. 144–161, 2010.

[28] J. Luo, K. D. Bowers, A. Oprea, and L. Xu, "Efficient software implementations of large finite fields gf (2 n) for secure storage applications," *ACM Transactions on Storage (TOS)*, vol. 8, no. 1, p. 2, 2012.

[29] J. Guajardo, S. Kumar, C. Paar, and J. Pelzl, "Efficient software- implementation of finite fields with applications to cryptography," *Acta Applicandae Mathematica*, vol. 93, no. 1, pp. 3–32.

[30] A. Reyhani-Masoleh, "Efficient algorithms and architectures for field multiplication using gaussian normal bases," *Computers, IEEE Transactions on*, vol. 55, no. 1, pp. 34–47, 2006.

[31] J. López and R. Dahab, "Fast multiplication on elliptic curves over gf (2m) without precomputation," in *Cryptographic Hardware and Embedded Systems*. Springer, 1999, pp. 316–327.

[32] G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "An implementation of elliptic curve cryptosystems over f 2 155," *Selected Areas in Communications, IEEE Journal on*, vol. 11, no. 5, pp. 804–813, 1993.

[33] A. Higuchi and N. Takagi, "A fast addition algorithm for elliptic curve arithmetic in gf (2 n) using projective coordinates," *Information processing letters*, vol. 76, no. 3, pp. 101–103, 2000.

[34] J. López and R. Dahab, "Improved algorithms for elliptic curve arithmetic in gf (2n)," in *Selected areas in cryptography*. Springer, 1999, pp. 201–212.

[35] D. Hankerson, J. L. Hernandez, and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields," in *Cryptographic Hardware and Embedded SystemsCHES 2000*. Springer, 2000, pp. 1–24.

[36] D. J. Bernstein and T. Lange, "Analysis and optimization of elliptic-curve single-scalar multiplication," *Contemporary Mathematics*, vol. 461, pp. 1–20, 2008.

[37] A. M. Fiskiran and R. B. Lee, "Evaluating instruction set extensions for fast arithmetic on binary finite fields," in *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*. IEEE, 2004, pp. 125–136.

[38] J. Groszschaedl and G.-A. Kamendje, "Instruction set extension for fast elliptic curve cryptography over binary finite fields gf (2 m)," in *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*. IEEE, 2003, pp. 455–468.

[39] G. Meurice de Dormale and J.-J. Quisquater, "High-speed hardware implementations of elliptic curve cryptography: A survey," *Journal of systems architecture*, vol. 53, no. 2, pp. 72–84, 2007.

[40] R. Bilal and M. Rajaram, "Design and implementation of high performance ecc co-processor," *International Journal of Engineering Science*, vol. 2, 2010.

[41] D. M. Schinianakis, A. P. Fournaris, H. E. Michail, A. P. Kakarountas, and T. Stouraitis, "An rns implementation of an f p elliptic curve point multiplier," *IEEE Transactions on Circuits and Systems Part I: Regular Papers*, vol. 56, no. 6, pp. 1202–1213, 2009.

[42] T. Wollinger, J. Pelzl, V. Wittelsberger, C. Paar, G. Saldamli, and Ç. K. Koç, "Elliptic and hyperelliptic curves on embedded μp," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 3, pp. 509–533, 2004.

[43] R. Petrusha, "Biginteger structure," 2012, http://msdn.microsoft.com/en-us/library/system.numerics.biginteger.aspx.

[44] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics - Doklady*, vol. 7, pp. 595–596, 1963.

[45] B. Sunar, "A generalized method for constructing subquadratic complexity GF(2/sup k/) multipliers," *Computers, IEEE Transactions on*, vol. 53, no. 9, pp. 1097–1105, 2004, iD: 1.

[46] J. Adikari, M. A. Hasan, and C. Negre, "Towards faster and greener cryptoprocessor for eta pairing on supersingular elliptic curve over f21223," Center of Applied Cryptographic Research (CACR), Tech. Rep. 23, 2012.

[47] T. Wollinger and V. Kovtun, "Fast explicit formulae for genus 2 hyperelliptic curves using projective coordinates," in *2007 4th International Conference on Information Technology New Generations*, Escrypt, Germany. Los Alamitos, CA, USA: IEEE Comput. Soc, 2-4 April 2007 2007, p. 5 pp.

[48] H. Wu, "Low complexity bit-parallel finite field arithmetic using polynomial basis," in *Cryptographic Hardware and Embedded Systems*. Springer, 1999, pp. 280–291.

[49] S. Chisholm, "Taxonomy of cryptographic pairings," Calgary University, Tech. Rep., 2008, master's thesis.

[50] P. Barreto, S. Galbraith, C. O'Eigeartaigh, and M. Scott, "Efficient pairing computation on supersingular abelian varieties," *Designs, Codes and Cryptography*, vol. 42, no. 3, pp. 239–271, 2007.

[51] D. G. Cantor, "Computing in the jacobian of a hyperelliptic curve," *Math.Comp*, vol. 48, no. 177, pp. 95–101, 1987.

[52] N. Koblitz, "Hyperelliptic cryptosystems," *Journal of Cryptology*, vol. 1, no. 3, pp. 139–150.

[53] P. L. Jensen, "Hyperelliptic curves and their application to cryptography," 2004.

[54] R. Harley, "Fast arithmetic on genus 2 curves, for c source code and further explanatios." [Online]. Available: http://cristal.inria.fr/~harley/hyper

[55] *Handbook of elliptic and hyperelliptic curve cryptography.* Boca Raton: Boca Raton : Chapman & Hall/CRC, 2006, iD: vtug3592853; Includes bibliographical references (p. 737-775) and indexes.

[56] T. Lange, "Inversion-free arithmetic on genus 2 hyperelliptic curves," Cryptology ePrint Archive, Tech. Rep. 147, 2002. [Online]. Available: http://eprint.iacr.org

[57] ——, "Formulae for arithmetic on genus 2 hyperelliptic curves," *Applicable Algebra in Engineering, Communication and Computing*, vol. 15, no. 5, pp. 295–328, 2005.

[58] C. Costello and K. Lauter, "Group law computations on jacobians of hyperelliptic curves," in *18th International Conference on Selected Areas in Cryptography, SAC 2011*, vol. 7118 LNCS, Information Security Institute, Queensland University of Technology, GPO Box 2434, Brisbane, QLD 4001, Australia. Toronto, ON, Canada: Springer Verlag, 2012, pp. 92–117.

[59] P. Gaudry, "Fast genus 2 arithmetic based on theta functions," *Journal of Mathematical Cryptology JMC*, vol. 1, no. 3, pp. 243–265, 2007.

[60] J. Balakrishnan, J. Belding, S. Chisholm, K. Eisenträger, K. E. Stange, and E. Teske, "Pairings on hyperelliptic curves," *WIN-Women in Numbers: Research Directions in Number Theory, Fields Institute Communications*, vol. 60, pp. 87–120, 2009.

[61] "Polymul: Fast polynomial multiplication in c++," 2009. [Online]. Available: https://code.google.com/p/polymul/

[62] D. Harvey, "zn_poly," 2008. [Online]. Available: http://web.maths.unsw.edu.au/~davidharvey/code/zn_poly/

[63] W. Hart, "Flint: Fast library for number theory." [Online]. Available: http://www.flintlib.org/

[64] "The gnu mp bignum library." [Online]. Available: http://gmplib.org/

[65] "Ntl: A library for doing number theory." [Online]. Available: http://www.shoup.net/ntl/

[66] "The gnu privacy guard - gnupg.org." [Online]. Available: http://www.gnupg.org/

[67] W. Dai, "Crypto++." [Online]. Available: http://www.cryptopp.com/

[68] P. Gutmann, "Cryptlib." [Online]. Available: http://www.cs.auckland.ac.nz/~pgut001/cryptlib/

[69] "Libgcrypt - gnu project - free software foundation (fsf)." [Online]. Available: http://www.gnu.org/software/libgcrypt/

[70] M. Scott, "Miracl crypt sdk." [Online]. Available: https://certivox.com/solutions/miracl-crypto-sdk/

[71] "Lidia: A c++ library for computational number theory." [Online]. Available: http://www.cs.sunysb.edu/~algorith/implement/lidia/implement.shtml,http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/

[72] "Openssl: The open source toolkit for ssl/tls." [Online]. Available: http://www.openssl.org/

[73] "Pari-gp home." [Online]. Available: http://pari.math.u-bordeaux.fr/

[74] A. Abusharekh and K. Gaj, "Comparative analysis of software libraries for public key cryptography," *Software Performance Enhancement for Encryption and Decryption, SPEED*, pp. 11–12, 2007.