# Elliptic Curve Cryptography: Schoof's Algorithm on Fields of Characteristic Two

by

Suen Tak Tsung, Daniel

Thesis

Submitted to the Faculty of the Graduate School of

The Chinese University of Hong Kong

(Division of Computer Science and Engineering)

In partial fulfillment of the requirements

for the Degree of

Master of Science

July, 2000

# Abstract

Elliptic curves find their practical importance in the field of cryptography, where the associated inverse logarithmic problem on the underlying finite group has been believed to be intractable in most cases. In practice, it is important to identify those exceptions so that only cryptographically-strong curves are selected for real-life applications. It turns out that the group order is one of the crucial criteria in identifying these exceptions. However, determining this group order used to be a time-consuming operation, especially for curves over huge finite fields, which is the case in most cryptographic applications.

It is well-known that points on elliptic curves over finite fields form finite groups under the chord-tangent composition law. The problem of determining the group order, also known as the point counting problem, has been an active research area in the past ten years.

In this dissertation, the point counting problem will be introduced on elliptic curves over fields of characteristic two. In particular, the Schoof's algorithm, the first algorithm that drops the complexity of the problem from $O(q^{\frac{1}{4}+\epsilon})$ for every positive $\epsilon$, to $O(\log^8 q)$, where $q$ is the order of the underlying field, will be discussed with its implementation details. Several example curves and their group orders determined by our implementation will also be presented.

## ACKNOWLEDGMENTS

I wish to express my gratitude to my supervisor, Phillip Leong, for his excellent guidance. I also wish to thank Thomas Pfahler of Technische Universitat Darmstädt, for his support in my investigation of the LiDIA library package.

# Contents

**1 Introduction** **5**

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.2 Contribution . . . . . . . . . . . . . . . . . . . . . . . . 8

1.3 Previous Work . . . . . . . . . . . . . . . . . . . . . . . 9

**2 Elliptic Curves** **10**

2.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2 Fields of Characteristic Two . . . . . . . . . . . . . . . 14

2.2.1 Polynomial Bases: Field Operations . . . . . . 15

2.2.2 Normal Basis: Field Operations . . . . . . . . 16

2.3 Division Polynomials . . . . . . . . . . . . . . . . . . . 18

2.4 Group Order . . . . . . . . . . . . . . . . . . . . . . . 20

2.5 Frobenius Map . . . . . . . . . . . . . . . . . . . . . . 21

**3 Schoof's Algorithm** **22**

3.1 Chinese Remainder Theorem . . . . . . . . . . . . . . . 23

3.2 Frobenius Map Identity . . . . . . . . . . . . . . . . . . 23

3.3 Strategy . . . . . . . . . . . . . . . . . . . . . . . . . . 24

3.3.1 $t(mod\ 2)$ . . . . . . . . . . . . . . . . . . . . . . 24

3.3.2 $t(mod\ p)$ for $p > 2$ . . . . . . . . . . . . . . . . 24

3.4 Implementation . . . . . . . . . . . . . . . . . . . . . . . 29

3.4.1 Precision . . . . . . . . . . . . . . . . . . . . . . . 29

3.4.2 Primes . . . . . . . . . . . . . . . . . . . . . . . . 29

3.4.3 Legendre's Symbol . . . . . . . . . . . . . . . . . 31

3.4.4 Division Polynomials $f_p$ . . . . . . . . . . . . . 31

3.4.5 $x^q, x^{q^2}, y^q, y^{q^2}\ (mod\ f_p)$ . . . . . . . . . . . . 31

3.4.6 Organization of Computations . . . . . . . . . 33

3.5 Summary and Performance Suggestion . . . . . . . . 35

**4 Results 36**

4.1 Examples . . . . . . . . . . . . . . . . . . . . . . . . . 38

**5 Conclusions and Future Developments 43**

5.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . 43

5.2 Future Developments . . . . . . . . . . . . . . . . . . 44

**6 Source Code Listing 45**

**Bibliography 69**

# Chapter 1

# Introduction

## 1.1 Motivation

In the last decade, elliptic curves have been extensively studied. One of the major reasons is that they have important properties which contribute to the solution of Fermat's Last Theorem. Apart from their theoretical value, elliptic curves find their practical application in the field of cryptography. Elliptic Curve Cryptosystem(ECC) is believed to be a good replacement of RSA scheme as elliptic curves are good sources of abelian groups that can be utilized in public-key cryptosystem, such as the Diffie-Hellman key exchange. Most importantly, it is widely believed that, in general, there is no sub-exponential-time algorithm to solve the associated discrete logarithmic problem.

Discrete logarithmic problems can be illustrated in cryptosystems such as the Diffie-Hellman key-exchange: parties $A$ and $B$ need to communicate with each other through a common session key. Let's assume that they are computing in a common known group $G$ with an element $g$. $A$ picks a positive integer $a$ and computes $g^a$, and sends this to $B$. Similarly, $B$ does the same thing, but picking his/her own positive integer $b$, and sends $g^b$ to $A$. Now, $A$ can compute $g^{ab}$ by multiplying $a$ times the value of $g^b$, which was received from $B$. $B$ does a similar procedure to get $g^{ab}$. This quantity $g^{ab}$ will then be the common session key between the two parties. This method is secure as long as the inverse logarithmic problems are difficult to solve. In the above example, three inverse logarithmic problems are involved:

1. given $g^a$, determine the value of $a$,
2. given $g^b$, determine the value of $b$,
3. given $g^{ab}$, determine the value of $ab$.

The first two problems assure that $A$ and $B$ cannot tell one another's private key. The third problem prevents interceptors from knowing $ab$, and thereby, determine the correct value of $a$ and $b$. With the order of $G$ a huge integer, the inverse logarithmic problem is generally believed to be intractable.

ECC can be compared with the traditional RSA cryptosystem where the underlying integer factorization problem can be solved pseudo-exponentially, interestingly, through the Lenstra's elliptic curves factorization algorithm. Hence, to achieve the same level of security, ECC uses far fewer bits than the corresponding RSA scheme.

In order that ECC be practically useful in reality, we have to identify what makes "good" curves. A "good" curve possesses nice properties that allows the construction of a strong cryptosystem, i.e. properties that make the underlying discrete logarithmic problem intractable. It turns out that one of the major criteria is the order of the underlying finite group. It was shown that cryptographically-strong cryptosystems can be built with a non-supersingular elliptic curve over a finite field, whose group order is divisible by a large prime as described on p.101 of [6], which is usually hundreds of bits long in practice. The problem of point counting is the determination of this quantity given an elliptic curve over a finite field. The earliest efficient algorithm that solves the point counting problem is Schoof's Algorithm, though the more recent Schoof-Elkies-Atkin algorithm (SEA) is more superior in performance. The understanding of Schoof's algorithm serves to lay the foundation in understanding and investigating more advanced techniques such as the SEA algorithm.

## 1.2 Contribution

In this thesis, we shall review the theories behind general elliptic curves. Our focus will only be on important results, i.e., results that are vital in subsequent chapters. These results are then followed by the explanation of Schoof's algorithm in Chapter 3. Our implementation of Schoof's algorithm is described and the results presented.

The underlying field of ECC is usually $GF(p)$, galois field of prime characteristic $p$, or $GF(2^m)$, galois field with $2^m$ elements. It is the latter kind of fields that we shall be focusing on as arithmetics inside these fields can be done without worrying about the carry bit. As a result, efficient hardware can easily be built to speed up calculations [9]. The Schoof's algorithm that we shall be studying will assume the underlying field is of this even characteristic.

## 1.3 Previous Work

Since the introduction of the point counting problem, different algorithms have been proposed to solve it efficiently. The first successful candidate was the Baby-Step Giant-Step (BSGS) method proposed by Shanks and Mestre. With $q$ being the order of the underlying field, the algorithm's runtime complexity is $O(q^{\frac{1}{4}+\epsilon})$ for arbitrary small $\epsilon > 0$. For more information about BSGS algorithm, see p.104 of [6]. Then in 1985, the runtime complexity was reduced to $O(\log^8 q)$ by Schoof [13]. Schoof's algorithm is based on calculations with the torsion points of the elliptic curve. After the introduction of Schoof's algorithm, Elkies and Atkin jointly discovered improvements to the Schoof's original algorithm that makes point counting practical. It is Schoof's original algorithm that we shall be focusing on in this dissertation.

# Chapter 2

# Elliptic Curves

In this chapter, we shall review the definitions and major results on elliptic curves over general fields in Section 2.1. Starting from Section 2.2, all fields are assumed to be of characteristic two, we shall adapt the results in Section 2.1 to this kind of finite fields, and see how the general formula for elliptic curves can be simplified in this special case. The division polynomials will be introduced in Section 2.3, where the relationships between these polynomials and points on the corresponding elliptic curves are investigated. The last section introduces the Frobenius endomorphism and the notion of supersingularity.

## 2.1 Definition

**Definition 2.1.1** *Let $\mathbb{F}$ be a field, an elliptic curve $E$ over $\mathbb{F}$ is defined as the set of points $(x, y)$ on $\mathbb{F}^2$ satisfying the Weierstrass's equation,*

$$E: \quad y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

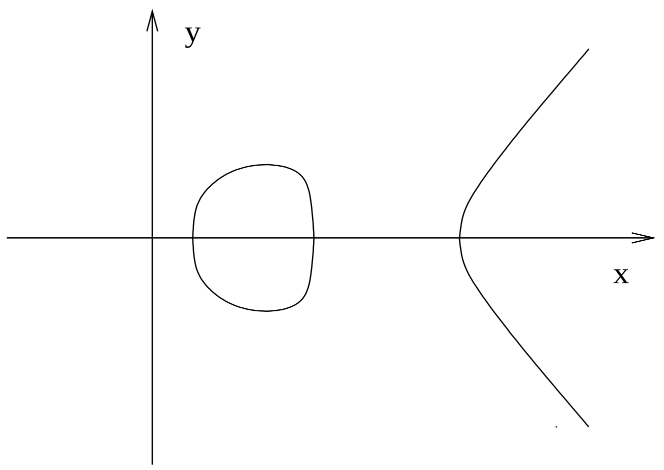
*where $a_i \in \mathbb{F}$ for $i \in \{1, 2, 3, 4, 6\}$. Moreover, for any point in $\overline{\mathbb{F}}^2$ on $E$, at least one of the followings are non-zero: $\frac{\partial F}{\partial x}$, $\frac{\partial F}{\partial y}$, where,*

$$F(x, y) = y^2 + a_1xy + a_3y - (x^3 + a_2x^2 + a_4x + a_6) = 0 \tag{2.2}$$

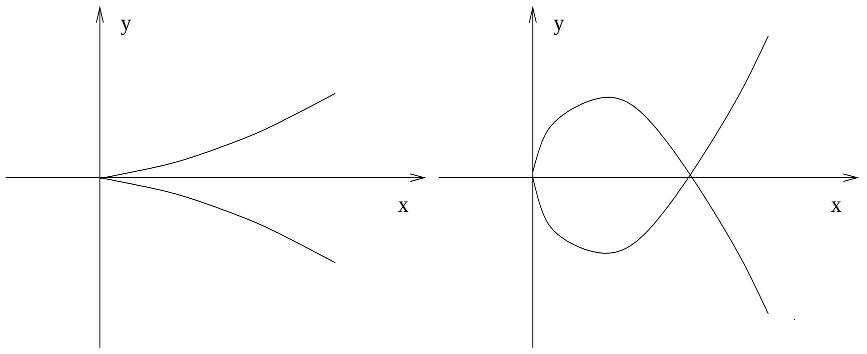
The field $\overline{\mathbb{F}}$ is an algebraic closure of $\mathbb{F}$, the field extension of $\mathbb{F}$ that every polynomials over $\overline{\mathbb{F}}$ factor completely into linear factors in $\overline{\mathbb{F}}$.

In this report, $\mathbb{F}$ will always denote a finite field, its characteristic is $char(\mathbb{F})$, and $E$ is always an elliptic curve over $\mathbb{F}$.

A point on a cubic curve is called a singular point if $\frac{\partial F}{\partial x} = \frac{\partial F}{\partial y} = 0$ at that point. A cubic curve is called a singular curve if it has a singular point. Otherwise, the curve is said to be non-singular. The following shows typical elliptic curves when $\mathbb{F} = \mathbb{R}$:



Clearly, elliptic curves are cubic algebraic curves with certain properties. The requirement on the partial derivatives not only serve to eliminate non-smooth curves, it also lets us avoid some undesirable situations. For example, a smooth curve may cross itself, thus giving two distinct tangents at the intersection point. This is illustrated below:

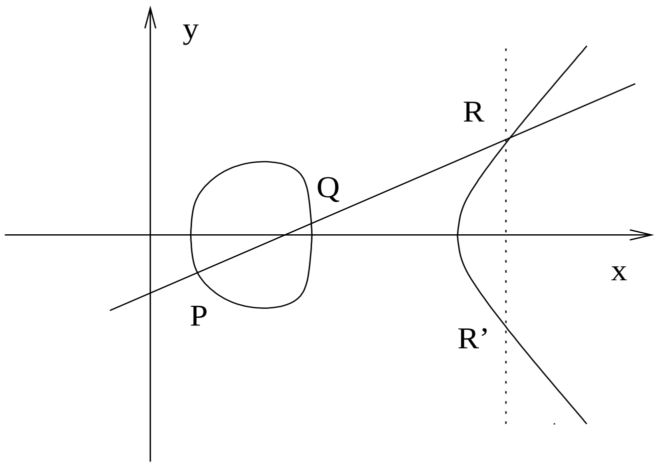


There are other quantities that are important as well. They are summarized as follow,

$$\begin{aligned}
d_2 &= a_1^2 + 4a_2 \\
d_4 &= 2a_4 + a_1a_3 \\
d_6 &= a_3^2 + 4a_6 \\
d_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \\
c_4 &= d_2^2 - 24d_4 \\
\Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\
j(E) &= \frac{c_4^3}{\Delta}
\end{aligned}$$

The quantity $\Delta$ is called the *discriminant* of the Weierstrass's equation and $j(E)$ is called the *j-invariant* of $E$.

Sometimes, it is useful to consider elliptic curves in projective plane, and in this case, in addition to points in $\mathbb{F}^2$, there is a point, denoted by $O$, which is commonly called the point at infinity. The introduction of such a point, together with the chord-tangent point composition law, forms a group, whose elements are points on the elliptic curve. The group operation is called point-addition, and the identity element is the point $O$. When $\mathbb{F} = \mathbb{R}$, this point composition law can be visualized geometrically as follow:



Draw a straight line through two points, $P$ and $Q$, on the elliptic curve, the third intersection point $R$ is taken as the "result" of "adding" points $P$ and $Q$. In fact, this is not yet a group, but if we do some operations on $R$ to get the point $R'$ that is still on the elliptic curve, this bulk of operations become a group operation. The figure above depicts an example of point addition just described. For detailed discussion on the group law, see [3] and [7].

Notice that when $\mathbb{F}$ is a finite group, this point addition is still a group operation as mentioned in [3].

We shall use the following notations: $E(\mathbb{F})$ denotes the set of points in $\mathbb{F}^2$ on $E$, together with the point $O$. If $E$ is defined over the field $\mathbb{F}$, the set of $\mathbb{F}$-rational points is $E(\mathbb{F})$.

## 2.2 Fields of Characteristic Two

In fields of characteristic two, one can consider only elliptic curves of the form,

$$E' : y^2 + xy = x^3 + a_2x^2 + a_6 \tag{2.3}$$

This is because there is an isomorphism between $E$ in (2.1) and $E'$ above through admissible change of variables. For details, please see p.16-17 of [2]. From now on, we shall be working solely in fields of characteristic two, and therefore, equation (2.3) is always assumed.

How are the associated quantities simplified to, such as $\Delta$, the discriminant, and $j(E)$, the $j$-invariant? It turns out that,

$$\Delta = a_6 \tag{2.4}$$
$$j(E) = \frac{1}{a_6} \tag{2.5}$$

See p.22 of [2].

As described in the last section, the points on an elliptic curve over finite fields form a group under the chord-tangent composition law. Below are explicit formulas for computing with this law in fields of characteristic two:

$$-P = (x, y + x) \tag{2.6}$$
$$\tag{2.7}$$

if $P = (x, y)$. If $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$, then their sum is $P_3 = (x_3, y_3)$ given by,

$$x_3 = \lambda^2 + \lambda + a_2 + x_1 + x_2$$
$$y_3 = (\lambda + 1)x_3 + x_3 + y_1$$

assuming that $P_3 \neq O$, and $\lambda$ is given by,

$$\lambda = \begin{cases} \frac{y_2+y_1}{x_2+x_1} & \text{if } x_1 \neq x_2 \\ \frac{x_1^2+y_1}{x_1} & \text{if } x_1 = x_2 \neq 0 \end{cases}$$

For more information on the these formulas, see p.37-38 of [6].

The most important observation is that if we denote the $x$-coordinate of a point $Q$ by $(Q)_X$, then, formula (2.6) implies,

$$(-P)_X = (P)_X \tag{2.8}$$

This is a useful fact in speeding up the algorithm as will be seen in the next chapter.

In fields of characteristic two, elements can be represented in two ways, namely, polynomial bases, and normal bases. In this report, we will use polynomial bases.

### 2.2.1 Polynomial Bases: Field Operations

It is well known that the quotient ring $\mathbb{F}[x]/\langle g(x)\rangle$ is a field if $g(x)$ is an irreducible element of $\mathbb{F}[x]$. What this means is that if $g(x) = r(x)t(x)$, where $r(x), t(x) \in \mathbb{F}[x]$, then either $r(x)$ or $t(x)$ is a constant. Let $\gamma$ be a root of $g(x)$ in some extension field of $\mathbb{F}$. Inside $\mathbb{F}[x]/\langle g(x)\rangle$, all elements can be regarded as polynomials of degree less than $n = deg(g(x))$, the degree of polynomial $g(x)$, in the element $\gamma$. In this case, the set $\{1, \gamma, \gamma^2, \ldots, \gamma^{n-1}\}$ is the basis for all elements in $\mathbb{F}[x]/\langle g(x)\rangle$. Elements in $\mathbb{F}[x]/\langle g(x)\rangle$ can therefore be represented using these bases.

In our implementation, we shall be doing arithmetic in the field $\mathbb{F}_2[x]/\langle g(x)\rangle$, for either a trinomial or pentanomial $g(x)$. These are "low-weight" polynomials that allow fast arithmetic operations. In particular, irreducible polynomials of the form $\gamma^m + \gamma^b + 1$ with large $m$ relative to $b$ is particularly interesting as efficient arithmetic operations can be implemented. See p.19-20 of [6] for details.

Addition and subtraction are the same carry-free operation, which can be done with bitwise XOR since $char(\mathbb{F}_2[x]/\langle g(x)\rangle) = 2$.

Algorithm 1 is duplicated from p.20 of [6] that performs modulo reduction. It assumes that we are multiplying two polynomials of degree $n-1$, and the result is divided by $g(x)$ to get the remainder in $\mathbb{F}_2[x]/\langle g(x)\rangle$.

**Algorithm 1** Reduction Modulo $g(x) = x^n + x^b + 1$, where $n > b > 0$

```
INPUT: $a(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{2n-2}x^{2n-2} \in \mathbb{F}_2[x]$
OUTPUT: $r(x) \equiv a(x)(mod\ g(x))$
**for** $i = 2n-2$ to $n$ step $-1$ **do**
  $a_{i-n} \leftarrow a_{i-n} + a_i$
  $a_{i-n+b} \leftarrow a_{i-n+b} + a_i$
RETURN $r(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$
```

Notice that modulo reduction is done in-place. Multiplication is done by normal multiplication followed by modulo reduction. A more efficient algorithm was based on recursive subdivision, first described by Karatsuba, which had the number of operations proportional to $O(n^{\log_2 3})$. The key observation is that, if $a(x), b(x) \in \mathbb{F}_2[x]$,

$$\begin{aligned}
a(x)b(x) &= (A_1(x)X + A_0(x))(B_1(x)X + B_0(x)) \\
&= [A_1(x)B_1(x)]X^2 + A_0(x)B_0(x) + [A_1(x)B_0(x) + A_0(x)B_1(x)]X \\
&= [A_1(x)B_1(x)]X^2 + A_0(x)B_0(x) \\
&+ [(A_0(x) + A_1(x))(B_0(x) + B_1(x)) - A_1(x)B_1(x) - A_0(x)B_0(x)]X
\end{aligned}$$

where $A_0, A_1, B_0, B_1$ are polynomials of degree $n/2 - 1$, and $X = x^{n/2}$. Notice that the following products need to be determined: $A_1(x)B_1(x), A_0(x)B_0(x)$, and $(A_0(x) + A_1(x))(B_0(x) + B_1(x))$. Therefore, a multiplication operation is divided into three multiplications of degree $n/2 - 1$. Karatsuba deployed this recursive relation in performing multiplications.

### 2.2.2 Normal Basis: Field Operations

A normal basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ is of the form $(\gamma, \gamma^2, \gamma^{2^2}, \ldots, \gamma^{2^{n-1}})$ for some $\gamma \in \mathbb{F}_{2^n}$. Such representation allows efficient hardware implementation as described by

Messey and Omura. [8]. Notice that squaring is just a simple bit shift operation. This allows fast repeat-squaring operation to be implemented in Schoof's algorithm, as will be seen in the next chapter. For multiplication, it has been shown that the number of operations involved is minimal if an optimal normal basis (ONB) is used. ONBs had been successfully identified. See [14] for details.

## 2.3 Division Polynomials

**Definition 2.3.1** *Let $E$ be an elliptic curve over $\mathbb{F}$, where $char(\mathbb{F}) = 2$, then we associate a set of division polynomials $f_n \in \mathbb{F}[x]$ to $E$ given by,*

$$\begin{array}{lcll}
f_0 & = & 0 & \\
f_1 & = & 1 & \\
f_2 & = & x & \\
f_3 & = & x^4 + x^3 + a_6 & \\
f_4 & = & x^6 + a_6 x^2 & \\
f_{2n+1} & = & f_n^3 f_{n+2} + f_{n-1} f_{n+1}^3 & n \geq 2 \\
f_{2n} & = & \frac{f_{n-1}^2 f_n f_{n+2} + f_{n-2} f_n f_{n+1}^2}{x} & n > 3
\end{array}$$

From now on, we shall always denote the $i$th division polynomial by $f_i$. In the point counting problem, the notion of torsion group is vital in our discussion in later sections.

**Definition 2.3.2** *The set of $m$-torsion group, denoted by $E[m]$, is given by,*

$$E[m] = \{P \in E(\overline{\mathbb{F}}) \mid mP = O\} \tag{2.9}$$

With the result below, the question on whether a given point is in $E[n]$ can easily be determined [6].

**Theorem 2.3.1** *Let $P$ be a point in $E(\overline{\mathbb{F}}) \setminus \{O\}$, and let $m \geq 1$. Then $P \in E[m]$ iff $f_m(P) = 0$.*

This theorem allows us to limit our attention on points in a specific torsion group when dealing with the Frobenius map identity as will be seen in chapter 3.

Division polynomials can also be used to compute the value of $nP$ without reference to the primitive point-addition formulas. This is summarized in the following result:

**Theorem 2.3.2** *Let $n \geq 2$, and let $P = (x, y) \in E \setminus \{O\}$ with $nP \neq O$. Then $nP = (\widetilde{x},\widetilde{y})$, where*

$$\widetilde{x} = x + \frac{f_{n-1}f_{n+1}}{f_n^2}$$

$$\widetilde{y} = x + y + \frac{f_{n-2}f_{n+1}^2}{xf_n^3} + (x^2 + x + y)\frac{f_{n-1}f_{n+1}}{xf_n^2}$$

## 2.4 Group Order

In 1983, Hasse has discovered the bounds of the group order. This is known as the Hasse's Theorem,

**Theorem 2.4.1** *Let $\mathbb{F} = \mathbb{F}_q$, then the number of $\mathbb{F}$-rational points on an elliptic curve $E$, denoted by $\#E(\mathbb{F})$, is given by,*

$$\#E(\mathbb{F}) = q + 1 - t$$

*where $-2\sqrt{q} \leq t \leq 2\sqrt{q}$.*

For a proof of the theorem, see [3] and [7].

With the Hasse's theorem, the point counting problem boils down to the determination of the value of $t$.

## 2.5 Frobenius Map

The Frobenius endomorphism $\varphi$ of a given curve $E$ over $\mathbb{F}_q$ is,

$$\varphi : \begin{cases} E(\overline{\mathbb{F}_q}) & \longrightarrow & E(\overline{\mathbb{F}_q}) \\ (x, y) & \longmapsto & (x^q, y^q) \\ O & \longmapsto & O \end{cases}$$

and for any point $P \in E(\overline{\mathbb{F}_q})$,

$$\varphi^2(P) - t\varphi(P) + qP = O \tag{2.10}$$

where $t$ is the same $t$ in Hasse's Theorem, which is called, the trace of Frobenius.

**Definition 2.5.1** *An elliptic curve $E$ over $\mathbb{F}$ is supersingular if $char(\mathbb{F}) \mid t$, the the trace of Frobenius.*

Supersingular curves are cryptographically weak curves and are therefore not suitable for practical use. See p.35 of [6]. It can be shown that,

**Theorem 2.5.1** *An elliptic curve $E$ over $\mathbb{F}$ is supersingular iff*

1. *$char(\mathbb{F}) = 2$ or $char(\mathbb{F}) = 3$, and $j(E) = 0$ or,*
2. *$char(\mathbb{F}) \geq 5$ and $t = 0$.*

Therefore, for curves of characteristic two, to guarantee non-supersingularity, $j(E) = \frac{a_1}{\Delta} \neq 0$, i.e., $a_1 \neq 0$.

The following theorem is duplicated from p.38 of [6],

**Theorem 2.5.2** *For an elliptic curve $E$ over $\mathbb{F}$ where $char(\mathbb{F}) = 2$,*

$$\#E(\mathbb{F}) \equiv 0 (mod\ 2) \tag{2.11}$$

As a result, using Hasse's Theorem, the trace of Frobenius $t$ must satisfy,

$$t \equiv 1 (mod\ 2) \tag{2.12}$$

# Chapter 3

# Schoof's Algorithm

In this section, the theory behind Schoof's algorithm will be presented followed by the implementation details.

## 3.1 Chinese Remainder Theorem

By Hasse's Theorem, for an elliptic curve $E$ over $\mathbb{F}_q$, $\#E(\mathbb{F}_q) = q + 1 - t$ where $|t| < 2\sqrt{q}$. Let $p_{max}$ be the smallest prime such that,

$$\prod_{p\ prime, 2 \leq p \leq p_{max}} p > 4\sqrt{q} \tag{3.1}$$

By the Chinese Remainder Theorem (CRT), the value of $t$ can be uniquely determined if $t(mod\ p)$ are known for all primes $p \in [2, p_{max}]$ [5].

How many primes do we need? If $n$ primes are required, by the Prime Number Theorem, we have,

$$n \sim O(\frac{p_{max}}{\log p_{max}}) \tag{3.2}$$

But, by p.140 of [1], $p_{max} \sim O(\log q)$ and therefore, combining equation (3.2) gives,

$$n \sim O(\frac{\log q}{\log \log q}) \tag{3.3}$$

Clearly, for a large value of $q$, i.e. a large finite field, the number of primes required to determine $t$ uniquely is still manageably small. This gives rise to opportunities in improving point counting algorithms.

## 3.2 Frobenius Map Identity

Recall that the Frobenius map $\varphi : (x, y) \mapsto (x^q, y^q)$ satisfying,

$$\varphi^2(P) - t\varphi(P) + qP = O \tag{3.4}$$

for any $P \in E(\overline{\mathbb{F}_q})$. We consider this equation for points in $E^*[p] = E[p] \setminus \{O\}$, the set of non-zero $p$-torsion points. Let $q_p \equiv q(mod\ p)$ where $q_p \in \{0, 1, ..., p-1\}$. If it can be shown that, for a point $P = (x, y) \in E^*[p]$,

$$(x^{q^2}, y^{q^2}) + q_p(x, y) = \tau(x^q, y^q) \tag{3.5}$$

for $\tau \in \{0, 1, ..., p-1\}$, then we must have $\tau \equiv t(mod\ p)$. Clearly, this $\tau$ is unique since $P \neq O$, and $p$ is a prime.

As mentioned in the last section, with these modulo quantities determined, $t$ can be uniquely recovered. Thus, the problem of point counting is reduced to the determination of $t(mod\ p)$ for all primes $p \in [2, p_{max}]$.

Notice that we are restricting ourselves to non-zero $p$-torsion points, this allows all computations be done modulo the division polynomials $f_p$'s as a result of theorem 2.3.1.

## 3.3 Strategy

### 3.3.1 $t(mod\ 2)$

In our implementation, we shall always assume that the curves interested are not supersingular. This means that $t \equiv 1(mod\ 2)$ as described in section 2.5.

### 3.3.2 $t(mod\ p)$ for $p > 2$

For each prime $3 \leq p \leq p_{max}$, we have to check which $\tau \in \{0, 1, ..., p-1\}$ satisfies the Frobenius map restricted to points in $E^*[p]$. First, we check if $\varphi^2 = -q_p$, i.e.,

$$(x^{q^2}, y^{q^2}) = -q_p(x, y) \tag{3.6}$$

This can be done by first equating the $x$-coordinates of both sides. Theorem 2.3.2 gives,

$$x^{q^2} = x + \frac{f_{q_p-1}f_{q_p+1}}{f_{q_p}^2} \tag{3.7}$$

Multiplying both sides by $f_{q_p}^2$ gives,

$$(x^{q^2} + x)f_{q_p}^2 + f_{q_p-1}f_{q_p+1} \equiv 0\ (mod\ f_p) \tag{3.8}$$

If equation (3.8) does not hold, $\varphi^2(P) \neq q_p$. Otherwise, there are two cases, (1)$\exists P \in E^*[p]$ with $\varphi^2(P) = -q_pP$, (2) $\exists P \in E^*[p]$ with $\varphi^2 = q_pP$. This is because $(P)_X = (-P)_X$. In the first case, $\varphi^2(P) + q_pP = O$, which implies that $\tau = 0$ since $\varphi(P) \neq O$. In the second case, we have $\varphi^2(P) = q_pP$, so that equation (3.5) simplifies to,

$$2q_p = \tau\varphi$$

Rearrange both sides, we get,

$$\varphi = \frac{2q_p}{\tau} \tag{3.9}$$

where $\tau^{-1}$ is the inverse of $\tau$ in $\mathbb{Z}_p$. Substitute equation (3.9) into equation (3.5) gives,

$$\frac{4q_p^2}{\tau^2} + q_p = 2q_p$$

which implies

$$\tau^2 = 4q_p \tag{3.10}$$

This means that $q_p$ has a square root in $\mathbb{Z}_p$.

With the above concept in mind, we first compute the Legendre's symbol $(\frac{q_p}{p})$. If $(\frac{q_p}{p}) \neq 1$, $q_p$ is not a quadratic residue modulo $p$, so that we must have $\varphi^2 = -q_p$, which implies that $\tau = 0$. Otherwise, let $\sigma^2 = q_p$ in $\mathbb{Z}_p$, then equation (3.10) gives,

$$\tau^2 = 4q_p = 4\sigma^2 \tag{3.11}$$

combining this result with equation (3.9), we have,

$$\varphi = \frac{2q_p}{\tau} = \frac{2q_p}{\pm 2\sigma} = \pm\frac{q_p}{\sigma} = \pm\sigma \tag{3.12}$$

To check for this condition, we equate the $x$-coordinates of both sides of equation (3.12), and eliminating the denominator, we get,

$$(x^q + x)f_\sigma^2 + f_{\sigma-1}f_{\sigma+1} \ (mod\ f_p) \tag{3.13}$$

If equation (3.13) does not hold, $\varphi^2 = -q_p$, so that $\tau = 0$. Otherwise, $\varphi^2 = q_p$, and we have to check if $\varphi = \sigma$ or $\varphi = -\sigma$. Equating the $y$-coordinates of both sides of (3.12) and eliminating the denominator, we get,

$$xf_\sigma^3(y^q + y) + f_{\sigma+2}f_{\sigma-1}^2 + (x^2 + y)f_{\sigma-1}f_\sigma f_{\sigma+1} \equiv 0 \ (mod \ f_p) \tag{3.14}$$

If this condition holds, by equation (3.9) and (3.10), $\varphi = \sigma$, and $\tau = \frac{2q_p}{\varphi} = \frac{2\sigma^2}{\sigma} = 2\sigma$, otherwise, $\varphi = -\sigma$, and $\tau = -2\sigma$.

At this stage, if $\tau$ cannot be determined in the steps above, then, $\tau \neq 0$, and we have to check each value of $\tau \in \{1, 2, ..., p-1\}$ in turn. As observed above, $(P)_X = (-P)_X$. By checking $\tau \in \{1, 2, ... \frac{p-1}{2}\}$ instead, the computations required are significantly reduced by half at this step.

This checking of $\tau$ requires equating the $x$-coordinates of both sides of equation (3.5) and see if equality holds. If so, we have found the appropriate $\tau$ and we can move on to the next prime, otherwise, we check for the next value of $\tau$. This results in the following two steps as described on p.138 of [1]:

1. Compute the following for each prime $p$,

$$\alpha = xf_{q_p}^3(y^{q^2} + y) + f_{q_p+2}f_{q_p-1}^2 + (x^2 + y)f_{q_p-1}f_{q_p}f_{q_p+1} \tag{3.15}$$

$$\beta = xf_{q_p}^3(x^{q^2} + x) + xf_{q_p-1}f_{q_p}f_{q_p+1} \tag{3.16}$$

$$g_\phi = f_{q_p}^2(((x^{q^2} + x)\beta + \alpha)\beta + \alpha^2) + \beta^2 f_{q_p-1}f_{q_p+1} \tag{3.17}$$

$$h_\phi = f_{q_p}^2(y^{q^2}\beta + x^{q^2}\alpha)\beta^2 + (\alpha + \beta)g_\phi \tag{3.18}$$

2. Repeat the following computation for each $\tau \in \{1, 2, ..., \frac{p-1}{2}\}$:

   - Compute,

$$f_\tau^{2q}g_\phi + \beta^2 f_{q_p}^2(f_\tau^{2q}x^q + f_{\tau-1}^q f_{\tau+1}^q) \ (mod \ f_p) \tag{3.19}$$

   This is the result of equating the $x$-coordinates of equation (3.5). If equation (3.19) does not evaluate to zero, we check for the next $\tau$.

Otherwise, we equate the corresponding $y$-coordinates, and compute,

$$\begin{aligned} x^q f_\tau^{3q} h_\phi \quad + \quad & f_{q_p}^2 \beta^3 (x^q f_\tau^{3q} y^q + f_{\tau+2}^q f_{\tau-1}^{2q}) \\ & + f_{q_p}^2 \beta^3 (x^{2q} + y^q) f_{\tau-1}^q f_\tau^q f_{\tau+1}^q \ (mod\ f_p) \end{aligned}$$

If the result is zero, $t \equiv \tau (mod\ p)$, otherwise, $t \equiv -\tau (mod\ p)$.

Finally, we are left with a set of $t(mod\ p)$ for enough $p$'s so that $t$ can be uniquely recovered in the range $[-2\sqrt{q}, 2\sqrt{q}]$ using the Chinese Remainder Theorem. The complete algorithm is shown on the next page.

**Algorithm 2** Schoof's Algorithm

```
 1: INPUT: $a_2, a_6$, and a finite field $\mathbb{F}$
 2: OUTPUT: $\#E(\mathbb{F})$
 3: determine primes used and stored in $p[]$, $n$ is the $\#$ of primes used
 4: compute division polynomials $f_i$ and stored in $f[i]$ for $i \in \{0, 1, ..., , n+1\}$
 5: compute $x^q, x^{q^2}, y^q, y^{q^2} (mod\ f[p[i]])$ for $i \in \{0, 1, ..., n\}$
 6: for $i = 1$ to $n$ do
 7:   $q_p \leftarrow q(mod\ p[i])$
 8:   if $f_{q_p}^2(x^{q^2} + x) + f_{q_p-1}f_{q_p+1} \equiv 0\ (mod\ f[p[i]])$ then
 9:     if $(\frac{q_p}{l}) = -1$ then
10:       $t[i] \leftarrow 0$
11:       BREAK
12:     else
13:       determine $\sigma \in \mathbb{Z}_{p[i]}$ such that $\sigma^2 = q_p$ by trial and error
14:       if $f_\sigma^2(x^q + x) + f_{\sigma-1}f_{\sigma+1} \not\equiv 0(mod\ f[p[i]])$ then
15:         $t[i] \leftarrow 0$
16:         BREAK
17:       else
18:         if $xf_\sigma^3(y^q + y) + f_{\sigma+2}f_{\sigma-1}^2 + (x^2 + y)f_{\sigma-1}f_\sigma f_{\sigma+1} \equiv 0\ (mod\ f[p[i]])$
            then
19:           $t[i] \leftarrow 2\sigma$
20:         else
21:           $t[i] \leftarrow -2\sigma$
22:   else
23:     $\alpha \leftarrow xf_{q_p}^3(y^{q^2} + y) + f_{q_p+2}f_{q_p-1}^2 + (x^2 + y)f_{q_p-1}f_{q_p+1}$
24:     $\beta \leftarrow xf_{q_p}^3(x^{q^2} + x) + xf_{q_p-1}f_{q_p}f_{q_p+1}$
25:     $g_\phi \leftarrow f_{q_p}^2(((x^{q^2} + x)\beta + \alpha)\beta + \alpha^2) + \beta^2 f_{q_p-1}f_{q_p+1}$
26:     $h_\phi \leftarrow f_{q_p}^2(y^{q^2}\beta + \alpha x^{q^2})\beta^2 + (\alpha + \beta)g_\phi$
27:     for $\tau = 1$ to $\frac{l-1}{2}$ do
28:       if $f_\tau^{2q}g_\phi + f_{q_p}^2\beta^2(x^q f_\tau^{2q} + f_{\tau-1}^q f_{\tau+1}^q) \equiv 0\ (mod\ f[p[i]])$ then
29:         $r \leftarrow f_2^q f_\tau^{3q} h_\phi + f_{q_p}^2\beta^3(x^q f_\tau^{3q} y^q + f_{\tau+2}^q f_{\tau-1}^q)$
30:         $r \leftarrow r + f_{q_p}^2\beta^3(x^{2q} + y^q)f_{\tau-1}^q f_\tau^q f_{\tau+1}^q (mod\ f[p[i]])$
31:         if $r = 0$ then
32:           $t[i] \leftarrow \tau$
33:         else
34:           $t[i] \leftarrow -\tau$
35: use $t[]$ and the Chinese Remainder Theorem to compute the unique $t$ in
    $[-2\sqrt{q}, 2\sqrt{q}]$
36: output $q + 1 - t$
```

## 3.4 Implementation

The Schoof's algorithm implemented assumes all computations be done in fields of characteristic two. As discussed in section 2.2, the general form of elliptic curve is thus given by $E : y^2+xy = x^3+a_2x^2+a_6$ where $a_2, a_6$ are input parameters. To specify the field that all computations be done, an irreducible polynomial must be supplied. Currently, only trinomials or pentanomials are supported. The elliptic curve is also assumed to be non-supersingular. The implementation used the Multiprecision Integer and Rational Arithmetic (MIRACL) C/C++ package by Shamus Software Ltd. to perform computations in $GF(2^m)$, as well as on large integers. For more information about MIRACL, please see [10].

### 3.4.1 Precision

The type that allows large integer arithmetic operation is the type `Big`. The number of bits used is set at compile time by a call to the function `mirsys(n, m)`, where `n` is the number of bits to be used. The types `Poly2` and `Poly2Mod` are used to capture elements in $\mathbb{F}_{2^m}$. They are internally represented by the type `Big`, and therefore, the same precision specified in `mirsys()` applies.

In our implementation, the value of `n` is pre-set to 600 to allow large finite groups. In practice, the underlying fields are picked from $GF(2^m)$s, where $m = 163, 233, 283, 409$, and $571$ [11].

### 3.4.2 Primes

Algorithm 3 on the next page returns the primes to be used given the order of the finite field $q$.

**Algorithm 3** Determine the Primes required in Schoof's Algorithm

```
INPUT: $q$, the order of the finite field $\mathbb{F}$
OUTPUT: $p[]$, array containing primes used in Schoof's Algorithm
$l \leftarrow 3$
$M \leftarrow 2$
$n \leftarrow 0$
**while** $M < 2\sqrt{q}$ **do**
  $M \leftarrow M * l$
  $l \leftarrow nextprime(l)$
  $n \leftarrow n + 1$
$n \leftarrow n + 1$ {allocate memory for $p[]$}
**for** $i = 0$ to $n - 1$ **do**
  $p[i] \leftarrow l$
  $l \leftarrow nextprime(l)$
RETURN $p[]$
```

### 3.4.3 Legendre's Symbol

The Legendre's symbol is computed with the algorithm cited on p.18 of [6]. The algorithm is duplicated as Algorithm 4 on the next page for easy reference to other implementors.

### 3.4.4 Division Polynomials $f_p$

Division polynomials are generated recursively from $f_0$ up to $f_{p_{max}}$ using the recursive formulas given in definition 2.3.1.

### 3.4.5 $x^q, x^{q^2}, y^q, y^{q^2}$ $(mod\ f_p)$

This is one of the trickiest part of the implementation. The desired powers are reached through repeat squaring. However, this is not straight-forward as squaring $y$'s results in high powers of $y$'s and high powers of $x$'s, that need to be resolved by substituting the corresponding expressions defined in $E$ for high powers of $y$'s. At every step in repeat squaring, this kind of substitutions is done so that the resulting expression is of the form $y^{2^i} = u_i(x) + v_i(x)y$ where $u_i$ and $v_i$ involve only terms in the variable $x$. Let's suppose that, at step 0, $i = 0$, we have

$$\begin{aligned} u_0(x) &= 0 \\ v_0(x) &= 1 \end{aligned}$$

At every step, we square the whole expression, so that, we have,

$$\begin{aligned} u_1(x) &= g(x) \\ v_1(x) &= x \end{aligned}$$

at the next step, where $g(x) = x^3 + a_2x^2 + a_6$. Squaring again gives,

$$\begin{aligned} u_2(x) &= g^2(x) + x^2g(x) \\ v_2(x) &= x^2 \times x = x^3 \end{aligned}$$

**Algorithm 4** compute $(\frac{a}{p})$, the Legendre's Symbol

```
INPUT: $a$ and $p$ integer
OUTPUT: $(\frac{a}{p})$
if $a \equiv 0(mod\ p)$ then
  RETURN 0
$x \leftarrow a$, $y \leftarrow p$, $L \leftarrow 1$
loop
  $x \leftarrow x(mod\ y)$
  if $x > y/2$ then
    $x \leftarrow y - x$
    if $y \equiv 3(mod\ 4)$ then
      $L \leftarrow -L$
  while $x \equiv 0(mod\ 4)$ do
    $x \leftarrow x/4$
  if $x \equiv 0(mod\ 2)$ then
    $x \leftarrow x/2$
    if $y \equiv \pm 3(mod\ 8)$ then
      $L \leftarrow -L$
  if $x = 1$ then
    RETURN L
  if $x \equiv 3(mod\ 4)$ and $y \equiv 3(mod\ 4)$ then
    $L \leftarrow -L$
  swap $x$ and $y$
```

Note that $v_{i+1}(x)$ is obtained from $v_i(x)$ by squaring $v_i(x)$, which is due to the previous repeat squaring procedure, and then multiply the result by $x$, which comes from $xy$ in the equation for $E$. Therefore, $v_i$ equals $1, x, x^3, x^7, ...$ at each repeat squaring step respectively. With this in mind, algorithm 5 on the next page computes $x^q, x^{q^2}, y^q, y^{q^2}$. The array $xtemp[]$ stores partial results when squaring $x$'s at each step, which is used to compute $v_i(x)'s$: at step $i$, in squaring $y$, an $x$ is multiplied to $xtemp[i]$ to give the correct value for $v_i(x)$.

### 3.4.6 Organization of Computations

It can be easily seen that careful organization of computations is vital to an efficient implementation. As seen in the Schoof's algorithm, the quantities $x^q, x^{q^2}$, $y^q, y^{q^2}$, as well as the products and powers of $f_{q_p}$'s and $f_\tau$'s, are seen everywhere, and it is important to compute them only once for all. This has been done in our implementation, so that repeat computation of the same quantity is avoided. In particular, the following quantities are to be computed once only for each iteration in the outermost loop: $f_{q_p-1}f_{q_p+1}$, $f_{\sigma-1}f_{\sigma+1}$, $x^{q^2}+x$. Also, inside the inner loop on the variable $\tau$, the quantity, $f^q_{\tau-1}f^q_{\tau+1}$ can be computed once and reused inside the same loop iteration.

**Algorithm 5** computes $x^q, x^{q^2}, y^q, y^{q^2}$

```
INPUT: $q = 2^m$, the field order; $g(x)$ where $E: \ y^2 + xy = g(x)$
OUTPUT: $x^q, x^{q^2}, y^q, y^{q^2}$ {compute $x^q$, store partial results for $y^q$, and $y^{q^2}$}
$xq \leftarrow 1$
for $k = 0$ to $m - 2$ do
    $xq \leftarrow xq \times x$
    $xq \leftarrow xq \times xq$
    $xtemp[k] \leftarrow xq$
STATE $xq \leftarrow xq \times x \times x$ {compute $y^q$}
$yqu \leftarrow g(x)$
for $k = 0$ to $m - 2$ do
    $yqu \leftarrow yqu \times yqu$
    $yqu \leftarrow yqu + (g(x) \times xtemp[k])$
$yqv \leftarrow x \times xtemp[m-2]$ {compute $x^{q^2}$ and $y^{q^2}$ together}
$xqtwo \leftarrow xtemp[m-2]$
$yqtwo \leftarrow yq$
for $k = 0$ to $m - 1$ do
    $xqtwo \leftarrow xqtwo \times x$
    $xqtwo \leftarrow xqtwo \times xqtwo$
    $yqtwo \leftarrow yqtwo \times yqtwo$
    $yqtwo \leftarrow yqtwo + (xqtwo \times g)$
$xqtwo \leftarrow xqtwo \times x \times x$
$yqtwou \leftarrow yqtwo$
$yqtwov \leftarrow xqtwo \times x$
{$x^q = xq$, $x^{q^2} = xqtwo$, $y^q = yqu(x) + yqv(x)y$,
$y^{q^2} = yqtwou(x) + yqtwov(x)y$}
```

## 3.5 Summary and Performance Suggestion

In general, computations can be done by assuming the curve $E$ is of the form,

$$E'' : \; y^2 + xy = x^3 + a_6 \tag{3.20}$$

instead of the general formula (2.3) as mentioned on p.102 of [2]. In fact, $E''$ and $E'$ of (2.3) are twist of one another if certain conditions hold. Twist curves are complement of one another in the following sense: $\#E'(\mathbb{F}_q) + \#E''(\mathbb{F}_q) = 2q + 2$ if $E'$ and $E''$ are twist of one another. This eliminates computations involving the $x^2$ term.

The bottleneck of the algorithm lies in the computations of the following quantities: $x^q, x^{q^2}, y^q, y^{q^2}$ $(mod\ f_p)$, and $f_\tau^q$ $(mod\ f_p)$. Similar time-consuming operations occur in evaluating powers of $f_\tau$.

In some cases, eigenvalue search can be deployed to avoid this time-consuming computation step all together as mentioned on p.105 of [2].

# Chapter 4

# Results

In this chapter, we shall show examples on point counting. Notice that the example curves are still not good enough to be practically applied in cryptography. As mentioned before, national standards require the power of 2 be at least 163 [11].

The examples are illustrated by running our implementation on an SMP machine with two PentiumIII 667MHz processors and 512MB of RAM. The runtimes required for all examples are summarized in table 4.1.

In all examples, all finite field elements are in polynomial representations in $t$. The value of $a_6$ will be shown in decimal, which is the sum of two to the powers

| **Example** | **Runtime (in seconds)** |
|---|---|
| 1 | 0.17 |
| 2 | 12.77 |
| 3 | 216.54 |
| 4 | 471.8 |

Table 4.1: Runtime Summary for Examples 1 to 4

| **Line Number(s)** | **Tag Reference** |
|---|---|
| 5-6 | [0] |
| 10 | [1] |
| 15 | [2] |
| 19, 21 | [3] |
| 32, 34 | [4] |

Table 4.2: Location Reference to Schoof's Algorithm

of $t$ in the corresponding field. For example, $a_6 = 15$ means that,

$$a_6 = t^3 + t^2 + t + 1$$

since $2^3 + 2^2 + 2^1 + 2^0 = 15$.

As can be seen in the list for the Schoof's algorithm in Section 3.3, the values of $\tau$ are determined at several locations, namely, line 10, 15, 19, 21, 32, and 34. We shall use table 4.2 to illustrate where the correct values of $\tau$ are picked up in all our examples.

For instance, let's suppose that in one of our examples, where $\tau \equiv t(mod\ p)$, and we find that $\tau = k$, and is picked up by Schoof's algorithm at line 21. This is represented as,

$$t \equiv k(mod\ p) \quad [3]$$

since the tag labeled "[3]" represents either line 19 or line 21. Notice that the tag "[0]" is used only for the case when $p = 2$. Since we are assuming that the input curve is non-supersingular, and therefore, $t \equiv 1(mod\ 2)$, i.e. $\tau = 1$. Hence, we do not need go through all lines below line 6 in this case. In our implementation, this assumption is made before the for-loop, i.e. at line 5-6 of the Schoof's algorithm listed.

We shall also step through example 1 in the next section as it is instructive in understanding Schoof's algorithm.

## 4.1 Examples

**Example 4.1.1**

$$\begin{array}{lll}
E & : & y^2 + xy = x^3 + 1 \\
\textit{Irreducible Polynomial} & : & x^6 + x^3 + 1 \\
\textit{Field} & : & \mathbb{F}_{2^6} \\
\textit{Primes used} & : & 2, 3, 5, 7 \\
\textit{Result} & : & t \equiv 1 (mod\ 2) \qquad [0] \\
& & t \equiv 0 (mod\ 3) \qquad [2] \\
& & t \equiv 4 (mod\ 5) \qquad [3] \\
& & t \equiv 2 (mod\ 7) \qquad [4] \\
& & \implies t = 9 \\
& & E(F_{2^6}) = 56
\end{array}$$

*For $p = 3$, $f_3 = x^4 + x^3 + 1$, the Schoof's algorithm goes as follow: check $(x^{q^2}, y^{q^2}) = -q_3(x, y)$. With $q_3 = 2^6 (mod\ 3) = 1$, equation (3.8) applies,*

$$\begin{aligned}
(x^{q^2} + x) f_1^2 + f_0 f_2 &= x^{64^2} + x \\
&\equiv x + x \\
&\equiv 0 (mod\ f_3)
\end{aligned}$$

*since $x^{64^2} \equiv x (mod\ f_3)$. Furthermore, the Legendre's symbol $(\frac{q_3}{l}) = (\frac{1}{3}) = 1$, and therefore, we have to pick a $\sigma \in \mathbb{Z}_3$ with $\sigma^2 = 1$. Clearly, $1^2 \equiv 1 (mod\ 3)$ and we check for equality of equation (3.13). Since,*

$$\begin{aligned}
(x^q + x) f_1^2 + f_0 f_2 &= x^{64} + x \\
&\equiv x^3 + x + 1 \\
&\not\equiv 0 (mod\ f_3)
\end{aligned}$$

*$\tau = 0$. So, the algorithm goes for the next value of $p$.*

*For* $p = 5$, $f_5 = \Sigma_{i\in\{0,3,4,5,6,8,10,11,12\}} x^i$ *we again check (3.8) with* $q_5 = 2^6 \equiv 4(mod\ 5)$,

$$\begin{aligned}(x^{q^2} + x)f_4^2 + f_3 f_5 &\equiv (x^{64^2} + x)f_4^2 \\ &\equiv 0(mod\ f_5)\end{aligned}$$

*Now,* $(\frac{q_5}{l}) = (\frac{4}{5}) = 1$. *We pick the quadratic residue* 2 *and compute equation (3.13):*

$$\begin{aligned}(x^q + x)f_2^2 + f_1 f_3 &= (x^{64} + x)f_2^2 + f_3 \\ &\equiv x^4 + x^3 + x + 1 + f_3 \\ &\equiv 0(mod\ f_5)\end{aligned}$$

*Now, we have to check equation (3.14),*

$$x f_2^3 (y^{64} + y) + f_4 f_1^2 + (x^2 + y) f_1 f_2 f_3 \equiv 0(mod\ f_5)$$

*and therefore* $\tau = 2 \times 2 = 4$.

*For* $p = 7$, $q_7 = 2^6(mod\ 7) = 1$. *Equation (3.8) gives,*

$$\begin{aligned}(x^{q^2} + x)f_1^2 + f_2 f_0 &\equiv x^{64^2} + x \\ &\equiv x^{23} + x^{19} + x^{18} + x^{12} + x^{11} + x^9 + x^8 + x^3 + x^2 + x + 1 \\ &\not\equiv 0(mod\ f_7)\end{aligned}$$

*therefore, we have to check for* $\tau \in \{1, 2, 3\}$. *First, we compute* $\alpha, \beta, g_\phi$, *and* $h_\phi$.

$$\begin{aligned}\alpha &= \Sigma_{\{i\in\mathbb{Z}\ \mid\ i\in\{0\leq i\leq 22\}\}\setminus\{0,8,11,18\}} x^i + y\Sigma_{j\in\{0,1,2,3,8,9,11,12,18,19,23\}} x^j \\ \beta &= \Sigma_{i\in\mathbb{Z}\ \mid\ i\in\{0\leq i\leq 22\}\setminus\{8,11,16,17,18\}} x^i \\ g_\phi &= \Sigma_{i\in\mathbb{Z}\ \mid\ i\in\{0\leq i\leq 23\}\setminus\{0,3,9,12,17,18,20\}} x^i \\ h_\phi &= x^7 + x^5 + x + y\Sigma_{i\in\{0,1,5,8,9,11,12,16,17,20,21,23\}} x^i\end{aligned}$$

*For $\tau = 1$, equation (3.19) gives,*

$$\begin{aligned} f_1^{128} g_\phi + \beta^2 f_1^2 (f_1^{128} x^{64} &+ f_0^{64} f_2^{64}) \\ &\equiv \Sigma_{\{i \in \mathbb{Z} \ | \ 0 \le i \le 23\} \setminus \{2,5,6,8,9,13,18,21\}} x^i \\ &\not\equiv 0 (mod\ f_7) \end{aligned}$$

*therefore, the next value of $\tau$ is tried. For $\tau = 2$, equation (3.19) gives,*

$$\begin{aligned} f_2^{128} g_\phi + \beta^2 f_1^2 (f_2^{128} x^{64} &+ f_1^{64} f_3^{64}) \\ &\equiv 0 (mod\ f_7) \end{aligned}$$

*and we have to check equation (3.20),*

$$\begin{aligned} x^{64} f_2^{192} h_\phi &+ f_1^2 \beta^3 (x^{64} f_2^{128} y^{64} + f_4^{64} f_1^{128}) \\ &+ f_1^2 \beta^3 (x^{128} + y^{64}) f_1^{64} f_2^{64} f_3^{64} \equiv 0 (mod\ f_7) \end{aligned}$$

*Therefore, $\tau = 2$ for $p = 7$.*

**Example 4.1.2**

| | | | |
|---|---|---|---|
| *E* | : | $y^2 + xy = x^3 + x^2 + 34$ | |
| *Irreducible Polynomial* | : | $x^{20} + x^3 + 1$ | |
| *Field* | : | $\mathbb{F}_{2^{20}}$ | |
| *Primes used* | : | $2, 3, 5, 7, 11, 13$ | |
| *Result* | : | $t \equiv 1 (mod\ 2)$ | *[0]* |
| | | $t \equiv 0 (mod\ 3)$ | *[2]* |
| | | $t \equiv 0 (mod\ 5)$ | *[2]* |
| | | $t \equiv 6 (mod\ 7)$ | *[4]* |
| | | $t \equiv 7 (mod\ 11)$ | *[4]* |
| | | $t \equiv 11 (mod\ 13)$ | *[4]* |
| | | $\implies t = -15$ | |
| | | $E(F_{2^{20}}) = 1048592$ | |

**Example 4.1.3**

| | | | |
|---|---|---|---|
| $E$ | : | $y^2 + xy = x^3 + x^2 + 11$ | |
| *Irreducible Polynomial* | : | $x^{41} + x^{40} + x^{39} + x^{38} + 1$ | |
| *Field* | : | $\mathbb{F}_{2^{41}}$ | |
| *Primes used* | : | $2, 3, 5, 7, 11, 13, 17, 19$ | |
| *Result* | : | $t \equiv 1 (mod\ 2)$ | *[0]* |
| | | $t \equiv 2 (mod\ 3)$ | *[4]* |
| | | $t \equiv 0 (mod\ 5)$ | *[1]* |
| | | $t \equiv 3 (mod\ 7)$ | *[4]* |
| | | $t \equiv 7 (mod\ 11)$ | *[4]* |
| | | $t \equiv 12 (mod\ 13)$ | *[4]* |
| | | $t \equiv 15 (mod\ 17)$ | *[4]* |
| | | $t \equiv 3 (mod\ 19)$ | *[4]* |
| | | $\Longrightarrow\ t = -2252485$ | |
| | | $E(F_{2^{41}}) = 2199025508038$ | |

**Example 4.1.4**

| | | | |
|---|---|---|---|
| $E$ | : | $y^2 + xy = x^3 + x^2 + 1234$ | |
| *Irreducible Polynomial* | : | $x^{53} + x^{13} + x^8 + x^3 + 1$ | |
| *Field* | : | $\mathbb{F}_{2^{53}}$ | |
| *Primes used* | : | $2, 3, 5, 7, 11, 13, 17, 19, 23, 29$ | |
| *Result* | : | $t \equiv 1(mod\ 2)$ | *[0]* |
| | | $t \equiv 2(mod\ 3)$ | *[4]* |
| | | $t \equiv 3(mod\ 5)$ | *[4]* |
| | | $t \equiv 5(mod\ 7)$ | *[4]* |
| | | $t \equiv 10(mod\ 11)$ | *[4]* |
| | | $t \equiv 2(mod\ 13)$ | *[4]* |
| | | $t \equiv 3(mod\ 17)$ | *[4]* |
| | | $t \equiv 1(mod\ 19)$ | *[4]* |
| | | $t \equiv 20(mod\ 23)$ | *[4]* |
| | | $t \equiv 9(mod\ 29)$ | *[4]* |
| | | $\Longrightarrow\ t = -130401217$ | |
| | | $E(F_{2^{53}}) = 9007199385142210$ | |

# Chapter 5

# Conclusions and Future Developments

## 5.1 Conclusions

In this thesis, the mathematics essential to the point counting problem has been reviewed. The Schoof's algorithm, and its implementation on fields of characteristic two, were described in great details and several examples have been included for illustration. The runtime were summarized and the last chapter contains the full source code of our implementation.

## 5.2 Future Developments

In our implementation, we use polynomial bases representation as our underling field. As was seen in chapter two, normal bases gives efficient hardware implementation.

The current state of art in point counting is the implementation of Elkies and Atkins' s improvements on the original Schoof's algorithm. Elkies and Atkin looked at the discriminant of the characteristic equation of Frobenius equation,

$$\delta_t = t^2 - 4q$$

If $\delta_t$ is a square modulo a prime $p$, then, the characteristic equation factors into two linear factors, and $p$ is called an Elkies prime. Otherwise, $p$ is called an Atkin prime. Notice that $\delta_t$ cannot be determined as $t$ is unknown. However, it turns out that this classification can be done in another manner through the splitting type of the $p$th modular polynomial. This polynomial also allows one to determine a factor of the $p$th division polynomial, which is of degree $(p-1)/2$. Reduction can be done through this factor instead of the division polynomial, and therefore, much time is saved in the computation. In fact, it has been shown on p.150 of [1] that their suggested improvements drop the complexity from Schoof's $O(\log^8 q)$ to $O(\log^3 q)$.

With the observation above, efficient point counting implementation will be employing Elkies and Atkin's improvements on Schoof's algorithm. Th elliptic curve selected will be over fields of characteristic two, whose elements are represented with normal bases, and computations be done in efficient hardware.

# Chapter 6

# Source Code Listing

This chapter contains the source code listing of the Schoof's algorithm implemented. The implementation is written in C/C++.

```
/*
 * Schoof's Algorithm on Elliptic Curve
 * E: Y^2 + XY = X^3 + A2 * X^2 + A6 over GF(2^m)
 * We shall follow [3] closely!
 * Daniel Suen (ttdsuen@ln.edu.hk), July 2000
 * Reference:
 * [1] Alfred J. Meenezes.
 *     Elliptic Curve Public Key Cryptosystems. Kluwer
 *     Academic Publishers.
 * [2] I. Blake, G. Seroussi, N. Smart.
 *     Elliptic Curves in Cryptography.
 *     Cambridge University Press.
 * [3] Andeas Enge.
 *     Elliptic Curves and their Applications to Cryptography,
 *     An Introduction. Kluwer Academic Publishers.
 */
```

```
#include <iostream>
#include "big.h"
#include "poly2.h"
#include "poly2mod.h"
#include "crt.h"
#include <string.h>
#include <ctype.h>

#define MAX(a, b) (((a) > (b)) ? (a) : (b))

extern bool init_prime_table (int, Big **, int &, Big, Big);
extern void init_ql_table (Big **, Big *, int, Big);
extern int legendre (int a, int p);
extern Big hextobig (char *buf);

// init_prime_table
// parameters:
//  m : 2^m, the order of the field
// primes : table of primes
// n : returns as the number of primes needed
// q : q = 2^m
// w : 4 * sqrt(q)
// this function determines the primes to be used in
// the Schoof's algorithm
// it also returns the number of primes used in the
// reference parameter n
// return value: true if things are ok
//   false if memoray allocation fails
```

```
bool
init_prime_table (int m, Big ** primes, int &n, Big q, Big w)
{
 Big l = 3;
 Big M = 2;
 n = 0;
 while (M < w)
   {
    M = M * l;
    l = nextprime (l);
    n++;
   }
 n++;
 *primes = new Big[n];
 int i;
 l = 2;
 if (*primes)
   {
    for (i = 0; i < n; i++)
      {
       (*primes)[i] = l;
       l = nextprime (l);
      }
    return true;
   }
 return false;
}

```

```


// init_ql_table
// parameters:
//  ql[] : table of q (mod primes[i])
// primes[] : table of primes
// m : 2^m, order of the field
// n : size of primes[]
// q : 2^m = q
// this function determines the all q (mod primes[i])
// and stored them in ql[]
void
init_ql_table (Big ** ql, Big * primes, int n, Big q)
{
 for (int i = 0; i < n; i++)
   {
    (*ql)[i] = q % primes[i];
   }
}

// legdendre
// parameters:
// a : the value to be tested if it's a
//          quadratic residue mod p
// p : the prime field characteristic
// return value: 1 if (a/p)=1, a is a quadratic
//                  residue mod p
//   0 if a = 0
//  -1 if (a/p)=-1, a is not a quadratic
//                   residue mod p
int
legendre (int a, int p)
{
 if (a % p == 0)
  return 0;
 int x = a;
 int y = p;
 int l = 1;
 while (1)
   {
     x = (x % y);
     if (x > (y / 2))
       {
         x = y - x;
         if (y % 4 == 3)
          l = -1;
       }
     while (x % 4 == 0)
      x = x / 4;
     if (x % 2 == 0)
       {
         x = x / 2;
         if (y % 8 == 3 || y % 8 == 5)
          l = -1;
       }
     if (x == 1)
     return l;
    if ((x % 4 == 3) && (y % 4 == 3))
     l = -l;
    int t = x;
    x = y;
    y = t;
   }
}

// hextobig
// paramters:
// buf : the character array that contains
//            the hex digits
// return value: the corresponding value in Big
Big
hextobig (char *buf)
{
 int len =::strlen (buf);
 Big retval = 0;
 int k = len - 1;
 while (k >= 0)
   {
    buf[k] =::tolower (buf[k]);
    int r = len - k - 1;
    if (isdigit (buf[k]))
     retval += pow ((Big) 16, r) * (buf[k] - '0');
    else
     retval += pow ((Big) 16, r) * (buf[k] - 'a' + 10);
     k--;
    }
 return retval;
}

int
main (int argc, char *argv[])
{
 miracl *mip = mirsys (600, 10);
 char torp;
 int m, a, b, c;
 char a6buf[256];
 int a2;
 Big a6;
 Big *ql;           // q mod l for each prime l
 Poly2 *f;          // division polynomial
 Poly2 *f2;         // division polynomial square
 Poly2 *f3;         // division polynomial cube
 Poly2 g;           // Y^2 + XY = g(X) = X^3 + A2 * X^2 + A6
 Poly2Mod *xqmod;   // x^q mod f[i] involving x
 Poly2Mod *xq2mod;  // x^(q^2) mod f[i] involving x
 Poly2Mod *yqmod;   // y^q mod f[i] involving x
 Poly2Mod *yqxymod; // y^q mod f[i] terms with y
 Poly2Mod *yq2mod;  // y^(q^2) mod f[i] terms involving x
 Poly2Mod *yq2xymod;// y^(q^2) mod f[i] terms involving y
 Big *tmodp;        // contains all t (mod p) for each p
 int j;
 bool noinput = true;
```

```

// get parameters to specify the field
// allow specification through an irreducible polynomial
cout << "Field Irreducible Polynomial" << endl;
cout << "===========================" << endl;
a = b = c = 0;
a6 = 0;
a2 = 0;
while (noinput)
  {
   cout << "\tTrinomial / Pentanomial [T/P]? ";
   cin >> torp;
   switch (torp)
     {
     case 'T':
      cout << "\tf(X) = X^m + X^a + 1" << endl;
      cout << "\tm = ";
      cin >> m;
      cout << "\ta = ";
      cin >> a;
      noinput = false;
      break;
     case 'P':
      cout << "\tf(X) = X^m + X^a + X^b + X^c + 1" << endl;
      cout << "\tm = ";
      cin >> m;
      cout << "\ta = ";
      cin >> a;
      cout << "\tb = ";
      cin >> b;
      cout << "\tc = ";
      cin >> c;
      noinput = false;
      break;
     default:
      cout << "invalid input, please try again" << endl;
     }
  }
// get parameters to specify the elliptic curve
cout << "Elliptic Curve ";
cout << "Y^2 + XY = X^3 + A2 * X^2 + A6" << endl;
cout << "===================================" << endl;
cout << "\tA2 = ";
cin >> a2;
char flag = 'N';
cout << "\tInput in Hex? [Y/N] ";
cin >> flag;
if (flag == 'Y')
  {
   cout << "\tA6 (non-zero) = ";
   cin >> a6buf;
   a6 = hextobig (a6buf);
  }
else
  {
   cout << "\tA6 (non-zero) = ";
```

```
    cin >> a6;
   }
// check that everything is ok
// (1) the field is correctly specified,
//     the polynomial specifed is
//     indeed an irreducible polynomial
// (2) check if the curve is not singular
if (!ecurve2 (m, a, b, c, a2, a6, TRUE, MR_AFFINE))
   {
    cout << "illegal curve parameters" << endl;
    cout << "m  = " << m << endl;
    cout << "a  = " << a << endl;
    cout << "b  = " << b << endl;
    cout << "c  = " << c << endl;
    cout << "a6 = " << a6 << endl;
    return 0;
   }
// we now let g = x^3 + a2 x^2 + a6
g = 0;
g.addterm (a6, 0);
g.addterm ((GF2m) 1, 3);
g.addterm (a2, 2);
cout << "E: Y^2 + XY = " << g << endl;
int n;
Big *primes = NULL;

// determine the number of primes to be used
Big q = pow ((Big) 2, m);
Big w = 4 * sqrt (q);
if (init_prime_table (m, &primes, n, q, w))
  {
   cout << "primes: ";
   for (int i = 0; i < n; i++)
     {
      cout << primes[i] << " ";
     }
   cout << endl;

   // computing division polynomials
   cout << "computing division polynomials..." << endl;
   int lastprime = toint (primes[n - 1]);
   ql = new Big[n];
   tmodp = new Big[n];
   init_ql_table (&ql, primes, n, q);
   f = new Poly2[MAX (lastprime, 5) + 1];
   f2 = new Poly2[MAX (lastprime, 5) + 1];
   f3 = new Poly2[MAX (lastprime, 5) + 1];
   xqmod = new Poly2Mod[MAX (lastprime, 5) + 1];
   xq2mod = new Poly2Mod[MAX (lastprime, 5) + 1];
   yqmod = new Poly2Mod[MAX (lastprime, 5) + 1];
   yqxymod = new Poly2Mod[MAX (lastprime, 5) + 1];
   yq2mod = new Poly2Mod[MAX (lastprime, 5) + 1];
   yq2xymod = new Poly2Mod[MAX (lastprime, 5) + 1];
   bool cond1 = !xqmod || !xq2mod || !yqmod;
   bool cond2 = !yqxymod || !yq2mod || !yq2xymod;
   if (cond1 || cond2)
```

```
    {
     cout << "cannot allocate memory for x^q, x^(q^2), ";
     cout << "y^q, y^(q^2)" << endl;
     return 0;
    }
  if (f && f2 && f3 && ql)
    {
     f[0] = 0;
     cout << "f[0]\t...done" << endl;
     f[1] = 1;
     cout << "f[1]\t...done" << endl;
     f[2] = 0;
     f[2].addterm (1, 1);
     cout << "f[2]\t...done" << endl;
     f[3] = 0;
     f[3].addterm (a6, 0);
     f[3].addterm (1, 3);
     f[3].addterm (1, 4);
     cout << "f[3]\t...done" << endl;
     f[4] = 0;
     f[4].addterm (a6, 2);
     f[4].addterm (1, 6);
     cout << "f[4]\t...done" << endl;
     f2[0] = 0;
     f3[0] = 0;
     f2[1] = 1;
     f3[1] = 1;
     f2[2] = 0;
```

```
      f2[2].addterm (1, 2);
      f3[2] = 0;
      f3[2].addterm (1, 3);
      f2[3] = f[3] * f[3];
      f3[3] = f2[3] * f[3];
      f2[4] = f[4] * f[4];
      f3[4] = f2[4] * f[4];
      for (int i = 5; i < lastprime + 1; i++)
        {
         cout << "f[" << i << "]\t...";
         if (i % 2 == 1)
           {
            j = (i - 1) / 2;
            f[i] = f[j + 2] * f3[j] + f[j - 1] * f3[j + 1];
           }
         else if (i % 2 == 0)
           {
            j = i / 2;
            Poly2 tt = f[j + 2] * f2[j - 1];
            tt += f[j - 2] * f2[j + 1];
            f[i] = f[j] * tt;
            f[i] = divxn (f[i], 1);
           }
         cout << "done" << endl;
         f2[i] = f[i] * f[i];
         f3[i] = f2[i] * f[i];
        }
     }
```

```
else
  {
   cout << "cannot allocate memory " << endl;
   cout << "for division polynomial" << endl;
   return 0;
  }
/*
 * computing x^q, x^(q^2), y^q, y^(q^2) mod f[i]
 * for each prime i
 */
Poly2 x;
Poly2Mod xq, yq, yq2, xq2;
Poly2Mod *xtemp = new Poly2Mod[m - 1];
for (int i = 1; i < n; i++)
  {
   int p = toint (primes[i]);
   setmod (f[p]);
   x = 0;
   x.addterm (1, 1);
   xq = 0;
   xq.addterm (1, 0);
   cout << "computing x^q mod f[" << p << "]" << endl;
   for (int k = 0; k < m - 1; k++)
     {
      xq = xq * x;
      xq = xq * xq;
      xtemp[k] = xq;
     }
```

```
      xq = xq * (x * x);
      xqmod[p] = xq;
      yq = g;
      cout << "computing y^q mod f[" << p << "]" << endl;
      for (int k = 0; k < m - 1; k++)
        {
         yq = yq * yq;
         yq = yq + (g * xtemp[k]);
        }
      yqmod[p] = yq;
      /*
       * needs to add one more x * xtemp[m-2] * y
       * for each yq
       */
      cout << "computing x^(q^2) mod f[" << p << "]" << endl;
      cout << "computing y^(q^2) mod f[" << p << "]" << endl;
      yqxymod[p] = x * xtemp[m - 2];
      xq2 = xtemp[m - 2];
      yq2 = yq;
      for (int k = 0; k < m; k++)
        {
         xq2 = xq2 * x;
         xq2 = xq2 * xq2;
         yq2 = yq2 * yq2;
         yq2 = yq2 + (xq2 * g);
        }
      xq2mod[p] = xq2 * x * x;
      yq2mod[p] = yq2;
      yq2xymod[p] = xq2 * x;
     }
   delete[]xtemp;
   /*
    * make tau = 1 (mod 2)
    * we are assuming that the curve is non-supersingular
    */
   tmodp[0] = 1;
   cout << "prime: 2\ttau = 1" << endl;
   /*
    * Schoof's Algorithm
    */
   for (int i = 1; i < n; i++)
     {
      int p = toint (primes[i]);
      int temp = (p - 1) / 2;
      int qlt = toint (ql[i]);
      /*
       * check if (x^(q^2), y^(q^2)) = [+|-] k (x, y),
       * i.e. Step 3 on p.137 of [3]
       */
      cout << "prime: " << p << '\t';
      setmod (f[p]);
      Poly2Mod x2 = x * x;
      Poly2Mod x3 = x2 * x;
      Poly2Mod r = (xq2mod[p] + x) * f2[qlt];
      Poly2 fqp1m1 = f[qlt - 1] * f[qlt + 1];
      r += fqp1m1;
```

```
        if (iszero (r))
          {
           // check the Legendre's symbol (ql[i] / primes[i])
           // if the result is -1, then ql is not a quadratic
           // residue of l, which implies that t = 0 (mod p)
           if (legendre (qlt, p) == -1)
             {
              tmodp[i] = 0;
              cout << "tau = " << tmodp[i];
             }
           else
             {
              // otherwise, by trial and error, find sigma with
              // sigma^2 = ql[i]. we check for all sigmas in
              // {0, 1, ..., p-1}
              for (int sig = 0; sig < p; sig++)
                {
                 int sig2modp = sig * sig;
                 sig2modp = sig2modp % p;
                 if (sig2modp == qlt)
                   {
                    // Then compute the fomula at the bottom
                    // of p.137 in [3].
                    r = f2[sig] * (xqmod[p] + x);
                    Poly2 fsigp1m1 = f[sig - 1] * f[sig + 1];
                    r += fsigp1m1;
                    if (!iszero (r))
                      {
```

```
                  tmodp[i] = 0;
                  cout << "tau = " << tmodp[i];
                  break;
                 }
               else
                 {
                  // else compute the formula
                  // at the top of
                  // p.138 , if it holds, then
                  // t = 2sigma(mod primes[i])
                  // else t = -2sigma(mod primes[i])
                  Poly2Mod fsigtrip = fsigp1m1 * f[sig];
                  Poly2Mod tempX = x * yqmod[p] * f3[sig];
                  tempX += f[sig + 2] * f2[sig - 1];
                  tempX += x2 * fsigtrip;
                  Poly2Mod tempY = x * f3[sig];
                  tempY = tempY + tempY * yqxymod[p];
                  tempY += fsigtrip;
                  if (iszero (tempX) && iszero (tempY))
                   tmodp[i] = 2 * sig;
                  else
                   tmodp[i] = -2 * sig + p;
                  cout << "tau = " << tmodp[i];
                  break;
                 }
              }             // if
           }                // for
        }                   // else
```

```
          }
        else
          {
           // (x^(q^2), y^(q^2)) != [+|-] k (x, y),
           // so we check for
           // all other possible values of tau, i.e. tau in
           // {1, 2, ..., (p-1)/2}, we check the sign later,
           // so that we actually cover
           // {-(p-1)/2,...,-1,0,1,...,(p-1)/2}
           // we are now at step 5
           // alpha = alphaX + alphaY * y
           // gphi = gphiX + gphiY * y
           // hphi = hphiX + hphiY * y
           Poly2Mod fqldoub = fqp1m1;
           Poly2Mod fqltrip = fqldoub * f[qlt];
           Poly2Mod alphaX = f3[qlt] * x;
           Poly2Mod beta = alphaX;
           Poly2Mod alphaY = alphaX;
           alphaX *= yq2mod[p];
           alphaX += f[qlt + 2] * f2[qlt - 1];
           alphaX += fqltrip * x2;
           alphaY = alphaY * yq2xymod[p] + alphaY;
           alphaY += fqltrip;
           Poly2Mod gphiX = xq2mod[p] + x;
           beta *= gphiX;
           gphiX += a2;
           beta += x * fqltrip;
           gphiX *= beta;
```

```
            gphiX += alphaX;
            gphiX *= beta;
            Poly2Mod alphaY2 = alphaY * alphaY;
            Poly2Mod beta2 = beta * beta;
            gphiX += alphaX * alphaX + alphaY2 * g;
            gphiX *= f2[qlt];
            gphiX += beta2 * fqldoub;
            Poly2Mod gphiY = alphaY * beta;
            gphiY += x * alphaY2;
            gphiY *= f2[qlt];
            Poly2Mod hphiX = yq2mod[p] * beta;
            hphiX += alphaX * xq2mod[p];
            Poly2Mod ttemp = f2[qlt] * beta2;
            hphiX *= ttemp;
            hphiX += beta * gphiX;
            hphiX += alphaX * gphiX;
            hphiX += alphaY * gphiY * g;
            Poly2Mod hphiY = yq2xymod[p] * beta;
            hphiY += alphaY * xq2mod[p];
            hphiY *= ttemp;
            hphiY += alphaX * gphiY;
            hphiY += alphaY * gphiX;
            hphiY += x * alphaY * gphiY;
            hphiY += beta * gphiY;
            // iterate for all possible values of tau
            for (int tau = 1; tau <= temp; tau++)
              {
               Poly2Mod f2qtau = f2[tau];
            Poly2Mod fqtau = f[tau];
            Poly2Mod fqtaupone = f[tau + 1];
            Poly2Mod fqtaumone = f[tau - 1];
            Poly2Mod fqtauptwo = f[tau + 2];
            for (int k = 0; k < m; k++)
              {
               fqtau = fqtau * fqtau;
               f2qtau = f2qtau * f2qtau;
               fqtaupone = fqtaupone * fqtaupone;
               fqtaumone = fqtaumone * fqtaumone;
               fqtauptwo = fqtauptwo * fqtauptwo;
              }
            ttemp = fqtaupone * fqtaumone;
            Poly2Mod tempX = f2qtau * gphiX;
            Poly2Mod rr = xqmod[p] * f2qtau + ttemp;
            tempX += f2[qlt] * beta2 * rr;
            Poly2Mod tempY = f2qtau * gphiY;
            if (iszero (tempX) && iszero (tempY))
              {
               // now the x-coordinate matches
               // check the y-coordinate to
               // determine if we have tau or -tau
               Poly2Mod f3qtau = f2qtau * fqtau;
               Poly2Mod beta3 = beta2 * beta;
               Poly2Mod f2qtaumone = fqtaumone * fqtaumone;
               tempX = xqmod[p] * f3qtau * yqmod[p];
               tempX += fqtauptwo * f2qtaumone;
               tempX *= f2[qlt] * beta3;
               tempX += xqmod[p] * f3qtau * hphiX;
               ttemp = xqmod[p] * xqmod[p] + yqmod[p];
               Poly2Mod fqtautrip = fqtaumone * fqtau;
               fqtautrip *= fqtaupone;
               tempX += ttemp * f2[qlt] * beta3 * fqtautrip;
               tempY = xqmod[p] * f3qtau * hphiY;
               rr = f2[qlt] * xqmod[p] * f3qtau;
               rr *= yqxymod[p] * beta3;
               tempY += rr;
               Poly2Mod ww = yqxymod[p] * f2[qlt];
               ww *= fqtautrip * beta3;
               tempY += ww;
               if (iszero (tempX) && iszero (tempY))
                tmodp[i] = tau;
               else
                tmodp[i] = -tau + p;
               cout << "tau = " << tmodp[i];
               break;
              }
           }                   // foreach tau >= 1
        }
      cout << endl;
     }                         // foreach prime
  }
else
  {
   cout << "could not allocate prime table" << endl;
   return 0;
  }
// we got all modulos, now it's time to do CRT.
Crt tcrt (n, primes);
Big t = tcrt.eval (tmodp);
Big lb = -w / 2;
Big ub = w / 2;
Big prod = 1;
for (int i = 0; i < n; i++)
 prod *= primes[i];
if (t <= lb)
 t += prod;
else if (t >= ub)
 t -= prod;
// now, let's output the trace t
cout << "t = " << t << endl;
// by Hasse's theorem, total number of points equals q+1-t
cout << "#E(F_{2^" << m << "}) = " << q + 1 - t << endl;
delete[]primes;
delete[]tmodp;
delete[]f;
delete[]f2;
delete[]f3;
delete[]xqmod;
delete[]xq2mod;
delete[]yqmod;
delete[]ql;
delete[]yq2mod;
delete[]yqxymod;
```

```
 delete[]yq2xymod;
 return 0;
}
```

# Bibliography

[1] Andreas Enge, *Elliptic Curves and Their Applications to Cryptography, An Introduction*, Kluwer Academic Publishers. (1999).

[2] Alfred J. Menezes *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers. (1993).

[3] Dale Husemöller *Elliptic Curves*, Graduate Texts in Mathematics, Springer-Verlag. (1986).

[4] Eric Von York *Elliptic Curves Over Finite Fields*, George Mason University. (1992).

[5] Harold M. Stark *An Introduction to Number Theory*, MIT Press. (1998) p.72-73.

[6] Ian Blake, Gadiel Seroussi & Nigel Smart *Elliptic Curves in Cryptography*, London Mathematical Society Lecture Note Series 265, Cambridge University Press (1999).

[7] Joseph H. Silverman, John Tate *Rational Points on Elliptic Curves*, Undergraduate Texts in Mathematics, Springer-Verlag. (1992).

[8] J. Omura and J. Massey *Computational method and apparatus for finite field arithmetic*, U.S. Patent number 4,587,627, May 1986.

[9] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong *FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor*, Department of Computer Science and Engineering, The Chinese University of Hong Kong.

[10] Mike Scott *Multiprecision Integer Rational Arithmetics C/C++ Library Documentation*, ftp://ftp.compapp.dcu.ie/pub/crypto/manual.zip.

[11] National Institute of Standards and Technology (NIST) *Recommended Elliptic Curves for Federal Government Use*, Computer Security Resource Centre, NIST (July 1999).

[12] Neal Koblitz *Algebraic Aspects of Cryptography*, Algoirthms and Computation in Mathematics, Volume 3, Springer-Verlag. (1991).

[13] Par René Schoof *Counting Points on Elliptic Curves over Finite Fields*, Journal de Théorie des Nombres, de Bordeaux 7 . (1995) p.219-254.

[14] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson *Optimal normal bases in $GF(p^n)$*, Discrete Applied Mathematics, vol 22, pp. 149-161, 1988/89.