

René Schoof's Algorithm  
for Determining the Order of the Group of Points  
on an Elliptic Curve over a Finite Field

John J. McGee

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State  
University  
in partial fulfillment of the requirements for the degree of

Master of Science  
in  
Mathematics

Dr. Ezra Brown, Chair  
Dr. Charles Parry  
Dr. Michael Williams

April 25, 2006  
Blacksburg, Virginia

Keywords: Elliptic Curve, Schoof, Cryptography

---

# René Schoof's Algorithm for Determining the Order of the Group of Points on an Elliptic Curve over a Finite Field

John McGee

## ABSTRACT

Elliptic curves have a rich mathematical history dating back to Diophantus (c. 250 C.E.), who used a form of these cubic equations to find right triangles of integer area with rational sides. In more recent times the deep mathematics of elliptic curves was used by Andrew Wiles et. al., to construct a proof of Fermat's last theorem, a problem which challenged mathematicians for more than 300 years. In addition, elliptic curves over finite fields find practical application in the areas of cryptography and coding theory. For such problems, knowing the order of the group of points satisfying the elliptic curve equation is important to the security of these applications. In 1985 René Schoof published a paper [5] describing a polynomial time algorithm for solving this problem. In this thesis we explain some of the key mathematical principles that provide the basis for Schoof's method. We also present an implementation of Schoof's algorithm as a collection of *Mathematica* functions. The operation of each algorithm is illustrated by way of numerical examples.

## Table of Contents

Chapter 1 - Introduction .....	1
1.1 Background .....	1
1.2 When is $f(x,y)$ an Elliptic Curve? .....	3
1.3 Addition of Points on an Elliptic Curve .....	4
Example 1 - Elliptic Curve Point Addition .....	7
Chapter 2 - Arithmetic in $\mathbb{F}_p$ .....	8
2.1 Elliptic Curves over Finite Fields .....	8
2.2 The Euclidean Algorithm .....	9
2.3 The Extended Euclidean Algorithm .....	10
Example 2 - The Extended Euclidean Algorithm .....	11
2.4 Finding the modular inverse .....	11
Example 3 - Multiplicative Inverse (mod $p$ ) .....	11
2.5 Modular Exponentiation .....	12
Example 4 - Modular Exponentiation .....	12
2.6 Square roots modulo $p$ .....	13
2.7 Shanks-Tonelli Modular Square Root Algorithm .....	13
Example 5 - Computing Square Roots Modulo $p$ .....	14
2.8 The Chinese Remainder Theorem .....	15
Example 6 - Determining the Chinese Remainder .....	15
Chapter 3 - Arithmetic of Elliptic Curves over $\mathbb{F}_p$ .....	16
Example 7 - Arithmetic in $E(\mathbb{F}_p)$ .....	16
Chapter 4 - Computing the Order of the Group $\# E(\mathbb{F}_q)$ .....	18
4.1 A direct method of computing $\# E(\mathbb{F}_q)$ .....	18
4.2 Overview of Schoof's Algorithm .....	20
4.3 Hasse's Theorem .....	21
4.4 Reducing the problem to that for $E(\mathbb{F}_p)$ .....	22
4.5 Baby Step, Giant Step Method .....	23
Example 8 - Determining Group Order using Hasse's Theorem .....	23
Chapter 5 - Schoof's Algorithm Implementation .....	24
5.1 Computing $t \pmod{2}$ .....	25
5.2 Determining if $x^3 + Ax + B$ has a root in $\mathbb{F}_q$ .....	26
Example 9 - Computation of $t \pmod{2}$ .....	27
5.3 The Division Polynomials .....	28
5.4 How many division polynomials? .....	29

Example 10 - Computation of the Division Polynomials ....	30
5.5 Computing $nP$ with the Division Polynomials .....	31
5.6 The Frobenius Endomorphism .....	32
5.7 The Characteristic Equation of the Frobenius .....	33
5.8 Schoof's Algorithm: Case One .....	35
5.9 Schoof Equation (17) .....	36
5.10 Schoof Equation (18) .....	37
5.11 Schoof's Algorithm: Case Two .....	38
5.12 Schoof Equation (19x) .....	39
5.13 Schoof Equation (19y) .....	40
5.14 Schoof's Algorithm Summary .....	42
Chapter 6 - Results of Running Schoof's Algorithm .....	43
6.1 A Detailed Example .....	43
6.2 Other Experiments .....	44
6.3 Discussion of Results .....	45
Chapter 7 - Applications .....	46
7.1 The Elliptic Curve Discrete Log Problem .....	46
7.2 Anomalous Curves and the MOV attack .....	46
References .....	47
Appendix A - Dictionary of <i>Mathematica</i> Functions for Elliptic Curves ...	48
Appendix B - <i>Mathematica</i> Code for Our Elliptic Curves Functions .....	52
Number Theoretic Algorithms .....	52
Elliptic Curve Arithmetic Algorithms .....	56
Methods to Determine the Elliptic Curve Group Order .....	61
The Functions that Comprise Schoof's Algorithm .....	65

## List of Figures

Figure 1 - René Schoof .....	2
Figure 2 - Plot of the Elliptic Curve $y^2 = x^3 - 5x - 2$ .....	4
Figure 3 - Number of digits in $p$ vs. number of small primes .....	30

## List of Tables

Table 1 - Points for $y^2 = x^3 + 46x + 74$ over $\mathbb{F}_{97}$ .....	19
Table 2 - Results from Schoof's Algorithm .....	45

## Chapter 1 - Introduction

"In re mathematica ars propendi pluris facienda est quam solvendi"  
- Georg Cantor.

### ■ 1.1 Background

Consider the following cubic polynomial in  $x, y$  over the field of real numbers  $\mathbb{R}$ :

$$y^2 = x^3 + Ax + B. \quad (1)$$

Suppose further that the right hand side of equation (1) has distinct roots. Then the graph of this curve is called an elliptic curve. Elliptic curves have a rich mathematical history dating back to Diophantus (c. 200 C.E.), who used a form of these cubic equations to find right triangles of integer area with rational sides. In more recent times some deep mathematical properties of elliptic curves were used by Andrew Wiles et. al., to construct a proof of Fermat's last theorem, a problem that had challenged mathematicians for more than 300 years. The Birch-Swinnerton-Dyer conjecture, one of the Clay Math Institute's million dollar problems, is also a question about certain mathematical properties of elliptic curves.

In addition, elliptic curves over the finite field  $\mathbb{F}_q$  for some large integer  $q$ , find practical application in the areas of cryptography and coding theory. One example of this is the Massey-Omura encryption method which relies on the difficulty of solving the elliptic curve discrete logarithm problem for security. For such methods, knowing the order of the group of points satisfying (1) with coefficients and coordinates in  $\mathbb{F}_q$ , written as  $\#E(\mathbb{F}_q)$ , is very important because a poor choice of curve parameters can lead to a situation that gives a potential eavesdropper the ability to break the code in reasonable time.

In 1985 René Schoof ( Figure 1) published a paper entitled "Elliptic curves over finite fields and the computation of square roots mod  $p$ " [5]. His paper describes a polynomial time algorithm for determining  $\#E(\mathbb{F}_q)$ . Refinements to his method by Elkies and Atkin have resulted in computer algorithms capable of finding results for elliptic curves over fields with orders greater than  $10^{100}$ [3,6].

**Figure 1 - René Schoof**

The purpose of this thesis is to explain the mathematical basis for Schoof's algorithm and to provide a *Mathematica* reference implementation of it. In order to achieve this goal we first present some background on elliptic curves in the  $x$ - $y$  plane. In particular we will see that the set points on the curve have the structure of an algebraic group. In chapter 2 we review the finite field arithmetic needed to work with elliptic curves over finite fields. In chapter 3 we present some basic algorithms for arithmetic in the group of points on an elliptic curve over a finite field. Chapter 4 describes some methods for computing the elliptic curve group order, and includes an introduction to Schoof's algorithm. We present the details of Schoof's algorithm in chapter 5. For each algorithm we first give a mathematical justification for the method or provide a reference to such. Next we present numerical examples that illustrates the operation of the algorithm. In chapter 6 we present results of running Schoof's algorithm against various curves. We conclude in chapter 7 with some applications that motivate efficient solutions to the elliptic curve group order problem. Appendix A contains a listing of the *Mathematica* functions that implement these algorithms, and Appendix B presents the *Mathematica* code for their implementation.

## ■ 1.2 When is $f(x,y)$ an Elliptic Curve?

Much of the following discussion is based on material presented in Lawrence Washington's book "Elliptic Curves - Number Theory and Cryptography" [7]. An *elliptic curve* is the set of points satisfying a nonsingular cubic polynomial in two variables. If  $K$  is a field, then an elliptic curve can be specified as

$$E : \{(x, y) \in K \times K \mid f(x, y) \in F[x, y], f(x, y) = 0\}, \quad (2)$$

where  $f(x, y)$  is a particular nonsingular polynomial in  $x, y$  of degree 3. A polynomial is nonsingular if it has distinct roots. If the field  $K$  has characteristic other than 2, 3, then by a transformation of variables it can be shown that  $E$  has the same behavior as an elliptic curve of the form:

$$E : y^2 = x^3 + A x + B. \quad (3)$$

Equation (3) is called Weierstrass equation of an elliptic curve.

Suppose  $a, b, c$  are the roots of the right hand side of (3). Then

$$y^2 = (x - a)(x - b)(x - c) = x^3 - (a + b + c)x^2 + (ab + ac + bc)x - abc.$$

Since the coefficient of  $x^2$  is zero we must have  $a + b + c = 0$ . Suppose that (3) has a double root. Without loss of generality let the double root be  $a = b$ . Then  $2a + c = 0$  so that  $a = -c/2$ , and we have

$$y^2 = x^3 - \frac{3}{4} c^2 x - \frac{1}{4} c^3.$$

Put  $A = \frac{-3}{4} c^2$ ,  $B = \frac{-1}{4} c^3$  so that  $y^2 = x^3 + A x + B$  then

$$4 A^3 = -\frac{27}{16} c^6 = -27 B^2.$$

Hence if (3) has a double root then  $4 A^3 + 27 B^2 = 0$ . The contrapositive gives that (3) has distinct roots only if

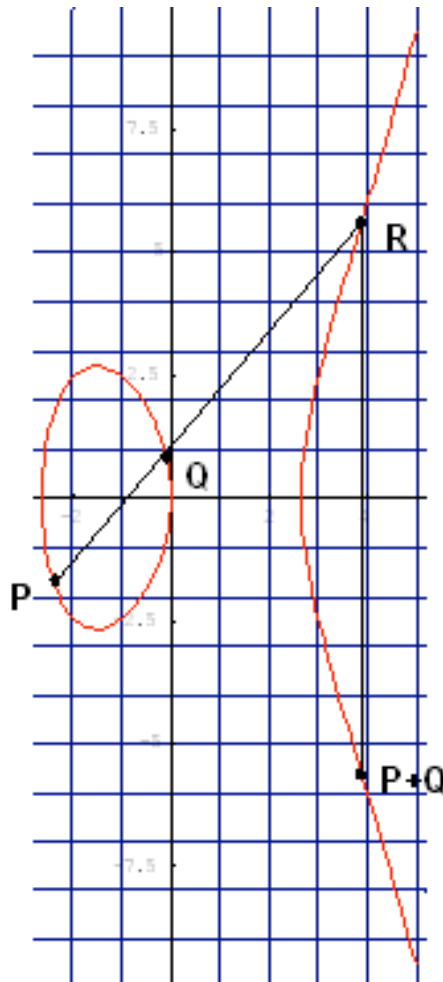
$$4 A^3 + 27 B^2 \neq 0. \quad (4)$$

The negative of the left hand side of (4) is called the *discriminant* of the elliptic curve.

### ■ 1.3 Addition of Points on an Elliptic Curve

For an elliptic curve  $E$ , take any two points  $P, Q$  that lie on  $E$ , then by Bezout's Theorem, the line between  $P$  and  $Q$  will intersect the curve  $E$  at a third point  $R$ . This is illustrated in Figure 2 for the elliptic curve given by  $y^2 = x^3 - 5x - 2$ . Notice that the line between two points would be vertical if the two points had the same  $x$ -coordinate. Such a line does not appear to intersect the curve. In this case we define the third point of intersection to be a special point at infinity,  $O$ . This definition is justified based on a consideration of the elliptic curve in projective coordinates, where  $O$  is a well-defined point.

Figure 2 - Plot of the Elliptic Curve  $y^2 = x^3 - 5x - 2$



Using this idea, we can define an addition operation on the elements of  $E$ . We define  $P + Q$  as follows. Let  $R$  be the third point of intersection of the line  $P - Q$  with the elliptic curve  $E$ . Then  $P + Q$  is the reflection of  $R$  about the line of sym-



metry of  $E$ , which is the  $x$ -axis for curves in Weierstrass form. If the line  $P - Q$  is vertical we define  $P + Q = O$ . To add  $P + P$ , we find the line tangent to the curve at  $P$  and intersect it with the curve to arrive at  $R$ .

With these definitions it can be shown that  $(E, +)$  actually forms an abelian group with identity  $O$ . That is, the operation is commutative, associative, and every point  $P$  has an inverse  $-P$  so that  $P + (-P) = O$ . In fact, for an elliptic curve in Weierstrass form if  $P = (x, y)$ , then  $-P = (x, -y)$ , the reflection of  $P$  about the  $x$ -axis. We have demonstrated that every element has an inverse and the proof of commutativity follows directly from the fact that the line through  $P, Q$  is the same as a line through  $Q, P$ . The proof of associativity is nontrivial. One method of proof is given in Washington [7] § 2.4

We can also define this addition operation using analytic geometry. Suppose that  $P = (x_1, y_1)$ ,  $Q = (x_2, y_2)$  are two points on the elliptic curve  $E: y^2 = x^3 + Ax + B$  with distinct  $x$ -coordinates. Then the slope of the line between the two points is given by

$$\lambda = (y_2 - y_1)/(x_2 - x_1). \quad (5)$$

The point-slope formula of the line passing through  $P, Q$  is given by

$$y - y_1 = \lambda(x - x_1). \quad (6)$$

Substituting (6) into the elliptic curve equation (3) yields

$$(\lambda(x - x_1) + y_1)^2 = x^3 + Ax + B.$$

Expanding and collecting terms in  $x$  gives the following monic polynomial in  $\mathbb{F}[x]$

$$\begin{aligned} x^3 - \lambda^2 x^2 + (2\lambda^2 x_1 - 2\lambda y_1 + A)x \\ + (B - \lambda^2 x_1^2 - y_1^2 + 2\lambda x_1 y_1) = 0. \end{aligned} \quad (7)$$

We know that  $x_1, x_2$  satisfy (7) because  $P$  and  $Q$  satisfy both the line and the elliptic curve equations. So we can factor the cubic (7) as

$$(x - x_1)(x - x_2)(x - x_3) = 0,$$

where  $x_3$  must be the  $x$ -coordinate of the third point of intersection. Expanding and collecting terms in  $x$  we obtain

$$x^3 - (x_1 + x_2 + x_3)x^2 + (x_1 x_2 + x_1 x_3 + x_2 x_3)x - x_1 x_2 x_3. \quad (8)$$

Because (7) and (8) represent the same polynomial, the coefficients of  $x^2$  must be equal, giving

$$-\lambda^2 = -(x_1 + x_2 + x_3).$$

Hence we can compute the  $x$ -coordinate of the third point of intersection as

$$x_3 = \lambda^2 - x_1 - x_2. \quad (9)$$

We can compute the corresponding  $y$ -coordinate using the equation of the line (6) and then negate the result to obtain the  $y$ -coordinate of  $P + Q$ ,

$$y_3 = -(\lambda(x_3 - x_1) + y_1) = \lambda(x_1 - x_3) - y_1. \quad (10)$$

On the other hand, if  $x_1 = x_2$  then  $y_1^2 = y_2^2$ , so either  $y_2 = y_1$  or  $y_2 = -y_1$ . If  $y_2 = -y_1$  then the line between  $P$  and  $Q_2$  is vertical, and we define  $P + Q = \mathcal{O}$ , the identity. Otherwise  $P = Q$ , so we want to compute  $P + P = 2P$ . To accomplish this we define  $\lambda$  as the slope of the tangent to the curve at  $(x_1, y_1)$ . We can compute this slope by implicit differentiation of (3) giving

$$2y dy = (3x^2 + A) dx$$

so that

$$\lambda = \frac{dy}{dx} = \frac{3x_1^2 + A}{2y_1}. \quad (11)$$

Using this  $\lambda$  with equations (9) and (10), along with the fact that  $x_1 = x_2$  and  $y_1 = y_2$  gives

$$(x_1, y_1) + (x_1, y_1) = (\lambda^2 - 2x_1, \lambda(x_1 - x_3) - y_1). \quad (12)$$

Equations (5) through (12) are incorporated into the *Mathematica* function **EcAdd**, which adds two points on an elliptic curve over  $\mathbb{R}$  or  $\mathbb{Q}$ .

**■ Example 1 - Elliptic Curve Point Addition**

Let an elliptic curve be given by  $E: y^2 = x^3 + 2x + 1$  over the rational numbers  $\mathbb{Q}$ . Then the discriminant is  $d = -4A^2 - 27B^3 = -4 * 4 - 27 = -43 \neq 0$ , so that  $E(\mathbb{Q})$  is a valid elliptic curve. It is easy to check, by substituting the point coordinates into the equation for  $E$ , that the points  $P = (0, 1)$  and  $Q = (1, -2)$  are elements of  $E(\mathbb{Q})$ . Also, the  $x$  coordinates of  $P, Q$  are distinct so we can compute the slope of the line passing through  $P, Q$  using (5) so that

$$\lambda = \frac{-2-1}{1-0} = -3.$$

Then equations (9) and (10) give

$$x_3 = (-3)^2 - 0 - 1 = 8 \text{ and } y_3 = -3(0 - 8) - 1 = 23.$$

So that  $P + Q = (8, 23)$ , which is also a point in  $E(\mathbb{Q})$  since  $8^3 + 2 * 8 + 1 = 529 = 23^2$ . We obtain the same result using the *Mathematica* function `EcAdd[{2,1},{0,1},{1,-2}]` which returns `{8, 23}`.

## Chapter 2 - Arithmetic in $\mathbb{F}_p$

### ■ 2.1 Elliptic Curves over Finite Fields

Suppose  $\mathbb{F}_q$  is a finite field of order  $q$ , then  $q = p^k$  for some integer  $k$  (see, for example Dummit and Foote [2] §14.3). Suppose also that  $E : y^2 = x^3 + Ax + B$  is an element of  $\mathbb{F}_q[x, y]$ , which means that  $A, B \in \mathbb{F}_q$ . Then, if  $4a^3 + 27b^2 \neq 0$  in  $\mathbb{F}_q$  it can be shown that

$$E(\mathbb{F}_q) = \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q \mid y^2 = x^3 + Ax + B\}$$

is an elliptic curve. Further, equations (5) through (12) of the previous section still obey the group axioms when arithmetic is done in the finite field, so that  $E(\mathbb{F}_q)$  is a finite abelian group. Since we have at most  $q$  choices for  $x$ , and for each of these at most 2 choices for  $y$ , then  $E(\mathbb{F}_q)$  contains at most  $2q$  points. It turns out that the actual bound is closer to  $q$ .

Before we consider this question in detail, we introduce some of the number-theoretic functions which are required for computation on elliptic curves over a finite field. For the purpose of our goal, which is to determine the group order  $\#E(\mathbb{F}_q)$ , it turns out that it will be sufficient to work with fields of prime order,  $\mathbb{F}_p$ . This allows us to perform arithmetic in the field using modular arithmetic in  $\mathbb{Z}/p\mathbb{Z}$ . That is, we can perform addition, subtraction and multiplication using ordinary integer arithmetic and then reduce each result mod  $p$  by dividing it by  $p$  and keeping the remainder.

Consider the following example in  $\mathbb{F}_7$ :

$$5 * 5 = 25 = 21 + 4 = 3 * 7 + 4 \equiv 4 \pmod{7},$$

so that in  $\mathbb{F}_7$ ,  $5 * 5 = 4$ .

Performing division in  $\mathbb{F}_p$  is a little bit more complicated. To compute  $a/b$  we need to multiply  $a$  by the modular inverse of  $b$ . For example, to compute  $4/5 \pmod{7}$  we must first find  $c$  such that  $5c \equiv 1 \pmod{7}$ , then  $4/5 = 4 * c$ . By trial and error we can find that  $5 * 3 = 15 \equiv 1 \pmod{7}$  so that  $4/5 \equiv 4 * 3 \equiv 5 \pmod{7}$ . We will later see how to use the Euclidean algorithm to

accomplish this for problems involving larger integers.

We will also have occasion to compute  $a^k \pmod{p}$ . This can be done by direct computation, for example  $3^4 = 81 \equiv 4 \pmod{7}$ , so that  $3^4 = 4$  in  $\mathbb{F}_7$ . Fortunately, for large  $k$  and  $p$ , there exists a much more efficient method based on doubling and reducing the result mod  $p$  at each step. The following sections describe these algorithms.

## ■ 2.2 The Euclidean Algorithm

The Euclidean Algorithm computes the greatest common divisor  $d$  of the integers  $a, b$ . It dates back to at least 300 B.C.E., where it appeared in geometric form in Euclid's *Elements*. The most obvious way to find the greatest common divisor is to completely factor both  $a$  and  $b$  into powers of primes such as

$$a = p_1^{r_1} p_2^{r_2} \dots p_n^{r_n} \text{ and } b = p_1^{s_1} p_2^{s_2} \dots p_n^{s_n},$$

where  $r_i$  and  $s_i$  are integers greater than or equal to zero and  $r_i = 0$  if  $p_i$  does not divide  $a$  and  $s_i = 0$  if  $p_i$  does not divide  $b$ . Then the greatest common divisor  $d$  is given by

$$d = p_1^{t_1} p_2^{t_2} \dots p_n^{t_n}, \text{ where } t_i = \min(r_i, s_i).$$

For example, if  $a = 7960 = 2^3 * 5 * 199$  and  $b = 6580 = 2^4 * 3^4 * 5$ , then  $\gcd(a, b) = 2^3 * 5 = 40$ .

However the integer factoring problem is currently understood to be difficult when the integer has no small prime divisors. The Euclidean algorithm is far superior because it can compute the greatest common divisor of two integers without factoring them. Suppose we wish to find the greatest common divisor of  $a$  and  $b$ , called  $\gcd(a, b)$ . Let  $a > b$  and divide  $a$  by  $b$  producing

$$a = s_0 b + r_0 \quad \text{where } 0 \leq r_0 < b \quad (\text{by the division algorithm}).$$

If  $r_0 = 0$ , then  $b$  divides  $a$  so  $\gcd(a, b) = b$ . Otherwise, divide  $b$  by  $r_0$  giving

$$b = s_1 r_0 + r_1 \quad \text{where } 0 \leq r_1 < r_0.$$

If  $r_1 = 0$  then  $r_0$  divides  $b$  and since  $a = s_0 b + r_0$  then  $r_0$  also divides  $a$  giving  $\gcd(a, b) = r_0$ . If  $r_i \neq 0$ , we can continue the process dividing  $r_{i-1}$  by  $r_i$  giving

$$r_0 = s_2 r_1 + r_2$$

...

$$r_{i-1} = s_{i+1} r_i + r_{i+1}.$$

Eventually we must find  $r_{n+1} = 0$  because at each step  $0 \leq r_i < r_{i-1}$ . Then  $r_{n-1} = s_{n+1} r_n$  so that  $\gcd(r_{n-1}, r_n) = r_n$ .

Note, however, that if  $x = qy + r$  then  $\gcd(x, y) = \gcd(y, r)$ . This is true because  $\gcd(y, r)$  divides both  $y$  and  $r$ , so it divides  $x$  also, hence  $\gcd(y, r)$  divides  $\gcd(x, y)$ . But we can write  $\gcd(y, r)$  as a linear combination of  $y$  and  $r$  so that

$$\gcd(y, r) = uy + vr = uy + v(x - qy) = (u - vq)y + vx,$$

so  $\gcd(x, y)$  divides  $\gcd(y, r)$ , hence  $\gcd(x, y) = \gcd(y, r)$ . Applying this to the chain of divisions above gives  $\gcd(r_{i-1}, r_i) = \gcd(r_{i-2}, r_{i-1})$ , so in particular  $\gcd(r_0, r_1) = \gcd(b, r_0) = \gcd(a, b)$ . Therefore the last nonzero divisor  $r_n = \gcd(a, b)$ . Euclid's algorithm for computing the greatest common divisor is implemented in the *Mathematica* function **EuclideanAlgorithm**.

### ■ 2.3 The Extended Euclidean Algorithm

The Extended Euclidean Algorithm computes the greatest common divisor  $d$  of the integers  $a, b$  and also computes two integers  $r, s$  such that  $d = ra + sb$ . This method provides a fast way to compute multiplicative inverses in  $\mathbb{F}_p$ . The algorithm proceeds as follows.

Starting with  $r_0 = 1, s_1 = 0$  we take  $d_0 = a = r_0 a + s_0 b$ . For step 1 we take  $r_1 = 0, s_1 = 1$  so we can write  $d_1 = b = r_1 a + s_1 b$ . At each succeeding step we compute the smallest positive  $d_i \equiv d_{i-2} \pmod{d_{i-1}}$ , so that  $d_i = d_{i-2} - k d_{i-1}$  for some positive integer  $k$ . The algorithm maintains  $d_i = r_i a + s_i b$  at each step so that

$$\begin{aligned} d_i &= (r_{i-2} a + s_{i-2} b) - k(r_{i-1} a + s_{i-1} b) \\ &= (r_{i-2} - k r_{i-1}) a + (s_{i-2} - k s_{i-1}) b. \end{aligned}$$

Hence, we must have  $r_i = r_{i-2} - k r_{i-1}$  and  $s_i = s_{i-2} - k s_{i-1}$ , which completes the

formulation of the recursion definition. Our *Mathematica* implementation is based on Rosen [4] §3.3.

■ **Example 2 - The Extended Euclidean Algorithm**

If we can find the prime factorization of two numbers then we can write down the greatest common divisor directly. It is the product of the largest prime powers that divide both numbers. For example, let  $a = 7960 = 2^3 * 5 * 199$  and  $b = 6580 = 2^4 * 3^4 * 5$ . Then  $\gcd(a, b) = 2^3 * 5 = 40$ . Even for such easy problems, however, the determination of  $\gcd(a, b)$  as a linear combination of  $a$  and  $b$  is best accomplished using the Extended Euclidean algorithm. Using `ExtendedEuclideanAlgorithm[ a, b ]` we find

$$\gcd(7960, 6480) = 40 = -35 * 7960 + 43 * 6480.$$

■ **2.4 Finding the modular inverse**

We can also use the Euclidean algorithm to find the modular inverse. The extended Euclidean algorithm finds the greatest common divisor of two numbers  $d = \gcd(a, b)$ . It also computes two numbers  $r, s$  such that  $d = r a + s b$ . If  $a$  is relatively prime to  $p$ , which is always true if  $p$  is a prime and  $1 < a < p$ , then  $\gcd(a, p) = 1$ . So to find the modular inverse of  $a$  modulo  $p$ , we use the Euclidean algorithm to compute

$$\gcd(a, p) = 1 = r a + s p \equiv r a \pmod{p}, \text{ hence } a^{-1} \equiv r \pmod{p}.$$

We need to compute modular inverses in order to perform the divisions in the elliptic curve point addition formulas.

■ **Example 3 - Multiplicative Inverse (mod  $p$ )**

As an example, let's work over  $\mathbb{F}_{19}$ , the field with 19 elements  $\{0, 1, \dots, 18\}$  with arithmetic modulo 19, a prime. In a field, every nonzero element has a multiplicative inverse, so let's find the inverse of 7. The Euclidean algorithm gives

$$\gcd(7, 19) = 1 = -8 * 7 + 3 * 19 \equiv -8 * 7 \pmod{19}.$$

But  $-8 \equiv 11 \pmod{19}$  so that  $11 * 7 \equiv 1 \pmod{19}$ . Hence  $7^{-1} \equiv 11 \pmod{19}$ . This is verified by the fact that  $11 * 7 = 77 = 4 * 19 + 1 \equiv 1 \pmod{19}$ .

## ■ 2.5 Modular Exponentiation

This method uses the binary representation of  $n$  to construct the result. Starts with  $a^1 \equiv a \pmod{p}$ ,  $x = 1$  then at each iteration  $k$  we compute

If the  $k^{\text{th}}$  bit of  $n$  is 1 then  $x \equiv x * a^k \pmod{p}$ .  
Then  $a^{2^k} \equiv a^k * a^k \pmod{p}$  for the next iteration.

In this way the arithmetic is done with relatively small integers, even though  $a^n$  may have hundreds or thousands of digits. In fact, at each step of the algorithm we multiply two numbers which are less than  $p$ , so the largest product we ever compute is less than  $p^2$ . Hence, if the binary representing of  $p$  requires  $m$  bits, then we need no more than  $2m$  bits to store the intermediate results. On the other hand, if we compute  $a^n$  directly, and  $a$  has  $m$  bits then we would need  $nm$  bits to hold the intermediate result.

### ■ Example 4 - Modular Exponentiation

As an example, let's compute  $11^{37} \pmod{97}$ . The direct method would first compute  $11^{37} = 340039485861577398992406882305761986971$  in  $\mathbb{Z}$  and then find  $11^{37} = 3505561709913169061777390539234659659 * 97 + 48$  so that  $11^{37} \equiv 48 \pmod{97}$ . However, in binary  $37 = 100101_2$ , so we can compute  $11^{37} \pmod{97}$  as follows.

$$11^{37} = 11^{32+4+1} = 11^{2^5+2^2+1} = 11^{(2^3+1)2^2+1} = \left( \left( \left( (11^2)^2 \right)^2 * 11 \right)^2 \right)^2 * 11.$$

But  $11^2 \equiv 24 \pmod{97}$ ,  $24^2 \equiv 91 \pmod{97}$  and  $91^2 \equiv 36 \pmod{97}$  so that  $11^{37} \equiv ((36 * 11)^2)^2 * 11 \pmod{97}$ . Then  $396 \equiv 8 \pmod{97}$ ,  $(8^2)^2 \equiv 22 \pmod{97}$ , hence  $11^{37} \equiv 22 * 11 \equiv 48 \pmod{97}$ .

Notice that we performed these computations without using any number larger than  $97^2$ . We will see later how this same idea, applied to polynomial arithmetic will allow us to compute  $f(x)^k \pmod{g(x)}$  in an efficient manner.



## ■ 2.6 Square roots modulo $p$

One of the steps in Schoof's algorithm requires the solution of the congruence  $w^2 \equiv a \pmod{p}$  for  $w$ , with  $p$  a prime number less than  $\sqrt{p}$ . In other words we need to find the square root of  $a$  modulo  $p$ . Unlike the multiplicative inverse problem, this problem does not always have a solution. When it does, we say that  $a$  is a quadratic residue modulo  $p$ , else  $a$  is called quadratic nonresidue. If there is an  $x$  such that  $x^2 \equiv a \pmod{p}$ , then  $a$  is a quadratic residue mod  $p$  and

$$a^{(p-1)/2} \equiv (x^2)^{(p-1)/2} \equiv x^{p-1} \equiv 1 \pmod{p} \text{ by Fermat's little theorem.}$$

Otherwise, for all  $i < p$ ,  $\gcd(i, p) = 1$ . Then for each  $i$  less than  $p$ ,  $i^2 \equiv a \pmod{p}$  has a unique solution, which can not be  $j = i$ , else  $i^2 \equiv a \pmod{p}$ . Hence we can group the solutions into  $(p-1)/2$  pairs, each with product congruent to  $a \pmod{p}$ . Taking the product of all of the solutions gives:

$$a^{(p-1)/2} \equiv (p-1)! \pmod{p},$$

since each number less than  $p$  is included exactly once in the product. Then Wilson's Theorem gives  $(p-1)! \equiv -1 \pmod{p}$ , so that

$$a^{(p-1)/2} \equiv -1 \pmod{p}.$$

Note that a more efficient algorithm exists which makes use of the Quadratic Reciprocity Theorem of Gauss, but we do not need the complexity of this method because we will be testing for quadratic residues for only modest sized integers.

## ■ 2.7 Shanks-Tonelli Modular Square Root Algorithm

Once we have determined that integer  $a$  is a quadratic residue modulo  $p$ , we need a method to find the square root. One method to accomplish this is called the Shanks-Tonelli modular square root algorithm. The details of the algorithm, which has performance logarithmic in the number of digits of  $p$ , are described in the paper "Square Roots from 1; 24, 51, 10 to Dan Shanks" by Ezra Brown [1].

■ **Example 5 - Computing Square Roots Modulo  $p$**

We consider the following nontrivial example. Let

$$p = 360027784083079948259017962255826129 .$$

We want to find  $x$  such that  $x^2 \equiv 2865 \pmod{p}$ . The Shanks-Tonelli algorithm gives

$$x = 203744876602447660339212047901408164 .$$

We could verify that this is the correct solution using the modular exponentiation method described above, but since we are only computing  $x^2 \pmod{p}$  we can compute this directly, showing that  $x^2 \equiv 2865 \pmod{p}$ .

## ■ 2.8 The Chinese Remainder Theorem

The Chinese Remainder Theorem provides a method to compute the smallest positive integer satisfying a set of congruences. It first appeared as a method of solution to a particular modular congruence problem in a third-century book by Chinese mathematician Sun Tzu [4] § 4.5. In Schoof's algorithm, we compute  $\tau_i \equiv t \pmod{l_i}$  for a set of small primes  $l_i$ , where  $t$  satisfies  $\#E(\mathbb{F}_p) = p + 1 - t$ . The Chinese Remainder Theorem allows us to recover  $t$  from this set of congruences, thus determining the order of the group.

We are given the following information, for the unknown  $z < N$

$$r_i \equiv z \pmod{n_i} \text{ for } i = 1, 2, \dots, k$$

where  $N = n_1 n_2 \dots n_k$  and  $\gcd(n_i, n_j) = 1$  when  $i \neq j$ , so that  $N$  is also the least common multiple of the  $\{n_i\}$ .

Let  $a_i = \frac{N}{n_i}$ ,  $b_i = a_i^{-1} \pmod{n_i}$ . The modular inverses  $b_i$  exist because  $n_i$  does not divide  $a_i$  because  $\gcd(n_i, n_j) = 1$  when  $i \neq j$ . Then  $a_i b_i \equiv 1 \pmod{n_i}$  and since  $n_j$  divides  $a_i$  when  $j \neq i$ , hence  $a_i b_i \equiv 0 \pmod{n_j}$  for  $j \neq i$ . So we can compute

$$z = \left( \sum_{i=1}^k a_i b_i r_i \right) \pmod{N},$$

which is the unique  $0 < z < N$  for which all the congruences hold.

### ■ Example 6 - Determining the Chinese Remainder

For example, suppose we have  $z \equiv 1 \pmod{2}$ ,  $z \equiv 0 \pmod{5}$ ,  $z \equiv 6 \pmod{7}$  and  $z \equiv 7 \pmod{11}$ . Then  $N = 770$  and the Chinese remainder algorithm gives

$$\begin{aligned} a_i &= \{385, 154, 110, 70\} \\ b_i &= a_i^{-1} \pmod{n_i} = \{1, 4, 3, 3\} \\ e_i &= a_i b_i = \{385, 616, 330, 210\} \\ z &= e \cdot r = 3835 \equiv 755 \pmod{N} \end{aligned}$$

So 755 is the smallest positive integer satisfying this set of congruences.

### Chapter 3 - Arithmetic of Elliptic Curves over $\mathbb{F}_p$

Now that we have a set of methods for performing arithmetic in  $\mathbb{F}_p$ , we can apply these to performing arithmetic in the group  $E(\mathbb{F}_p)$ . We provide a set of *Mathematica* functions to implement algorithms for testing if a cubic equation is an elliptic curve over  $\mathbb{F}_p$ , for adding points on the curve, and for efficiently computing  $kP$ , the sum of  $k$  copies of the point  $P$ . In each case it is assumed, and for efficiency sake not verified, that  $p$  is a prime.

■ **Example 7 - Arithmetic in  $E(\mathbb{F}_p)$**

We will now review several of the mathematical ideas and methods related to elliptic curves over a prime field by way of an example. Consider the elliptic curve  $E : y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$ . It has discriminant

$$d \equiv -(4a^3 + 27b^2) \equiv -(4 * 46^3 + 27 * 74^2) \equiv 537196 \equiv 87 \pmod{p}$$

Since the discriminant is nonzero modulo  $p$ ,  $E$  is a valid elliptic curve.

At  $x = 1$  we have

$$x^3 + 46x + 74 = 1 + 46 + 74 = 121 \equiv 24 \pmod{97}$$

so  $E$  has a point with  $x = 1$  if and only if 24 is a quadratic residue modulo 97. Using modular exponentiation we find with  $p = 97$  that  $24^{(p-1)/2} \equiv 1 \pmod{p}$ , so that 24 has a square root modulo  $p$ . We then employ the Shanks-Tonelli algorithm to find this square root giving  $11^2 \equiv 24 \pmod{p}$  so that the point  $P = (1, 11)$  is an element of  $E(\mathbb{F}_p)$ .

Next let us compute  $P + P = 2P$  using Equation (12) modulo  $p$  as

$$\lambda = \frac{3x^3 + A}{2y} \equiv (3 + 46)(22^{-1}) \pmod{97} \equiv 49 * 75 \pmod{97} \equiv 86 \pmod{97}.$$

Then

$$\begin{aligned} x_3 &= \lambda^2 - 2x_1 \equiv 86^2 - 2 \pmod{97} \equiv 22 \pmod{97}, \text{ also} \\ y_3 &= \lambda(x_1 - x_3) - y_1 = 86(1 - 22) - 11 \pmod{97} \equiv 26 \pmod{97}. \end{aligned}$$

Hence  $2 * (1, 11) = (22, 26)$  on  $E(\mathbb{F}_{97})$ . We can also compute  $4 * (1, 11)$  by add-

ing  $(22, 26) + (22, 26)$  giving  $(4, 15)$  or obtain the same result using the function `EcPowerMod[{46, 74}, {1, 11}, 4, 97]`.

The order of a point  $P \in E(\mathbb{F}_p)$  is defined as the smallest  $k$  such that  $kP = \mathcal{O}$ . One way to determine  $k$  is to compute  $nP$  for each successive  $n$  starting at 1 until the result is the identity. For the previous example with  $E: y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$  and  $P = (1, 11)$  we find, using repeated application of `EcAddMod`, that

$$2P = (2, 26), 3P = (27, 12), 4P = (4, 15), \dots, 15P = (1, 86).$$

Since  $86 + 11 \equiv 0 \pmod{97}$ , then  $15P = -P$ , so we know that  $16P = \mathcal{O}$ . Since 16 is the smallest multiple of  $P$  for which this occurs we have the order of  $P$  is 16 and we write  $|P| = 16$ . This gives a hint as to one way to determine  $\#E(\mathbb{F}_p)$ . By Lagrange's theorem the order of the element of a finite group must divide the order of the group. So we know that 16 divides  $\#E(\mathbb{F}_{97})$  for our sample curve.

## Chapter 4 - Computing the Order of the Group $\# E(\mathbb{F}_q)$

### ■ 4.1 A direct method of computing $\# E(\mathbb{F}_q)$

A direct approach to determining  $\# E(\mathbb{F}_q)$  is to compute  $z = x^3 + Ax + B$  for each  $x \in \mathbb{F}_q$ , and then to test if  $z$  has a square root in  $\mathbb{F}_q$ . If  $z = 0$ , then  $(x, 0) \in E(\mathbb{F}_q)$ . If there exists  $y \in \mathbb{F}_q$  such that  $y^2 = z$ , then  $(x, y), (x, -y) \in E(\mathbb{F}_q)$ , else there is no point in  $E(\mathbb{F}_q)$  with  $x$ -coordinate  $x$ . This means there are at most  $2q + 1$  elements in the group. However, a theorem of finite fields states that exactly  $1/2$  of the non-zero elements of  $\mathbb{F}_q$  are quadratic residues. This means that, on average, there will be approximately  $q + 1$  elements in  $E(\mathbb{F}_q)$ . As we shall see next, specific bounds on  $\# E(\mathbb{F}_q)$  can be established, and this is one key to more efficient methods of determining the group order.

Before proceeding, we want to fully characterize all of the points in  $E(\mathbb{F}_p)$  for our example curve  $E : y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$ . We can do this using the *Mathematica* function `FindEcPointSet` which encodes the technique above to find every point on the curve. Then for each point we can determine its order using the technique outlined at the end of chapter 3. This method is encoded in the *Mathematica* function `EcPointOrderMod`. The results of applying these methods to our example are shown in the table of Figure 3. Note that for each  $x$  there are two values of  $y$ , which are distinct unless  $y = 0$ . This is so because each solution has  $y^2 \equiv z \pmod{p}$  where  $z = x^3 + Ax + B$  for a particular value of  $x$ , so if  $y_1^2 = z$  then  $(-y_1)^2 = z$ . There can be no other distinct solutions because the quadratic equation  $y^2 - z = 0$  can have only two solutions. Observe that for each pair of points in Figure 3 with the same value of  $x$  we have  $y_2 \equiv -y_1 \pmod{p}$ , so that the points  $(x, y_1), (x, y_2)$  are indeed inverses in  $E(\mathbb{F}_p)$ .

If we count these points we find 39 pairs of points  $(x, y_1), (x, y_2)$  where  $y_1 \neq y_2$ . We also have one point,  $(57, 0)$ , with  $y = 0$ . Including the identity  $O$ , there are a total of  $2 * 39 + 1 + 1 = 80$  points so that  $\# E(\mathbb{F}_p) = 80$ . The last column in the table gives the order of each point. Notice also that each point order divides 80, the order of the group, as must be so by Lagrange's theorem.

**Table 1 - Points for  $y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$** 

x	y <sub>1</sub>	y <sub>2</sub>	Ord (P)
1	11	86	16
4	15	82	4
6	9	88	80
8	9	88	40
9	21	76	10
10	46	51	8
15	29	68	80
19	12	85	80
20	19	78	80
22	26	71	8
24	8	89	40
27	12	85	16
30	18	79	40
32	48	49	40
34	28	69	80
35	6	91	20
37	7	90	80
43	46	51	80
44	46	51	80
46	2	95	20
49	45	52	5
51	12	85	10
52	22	75	80
57	0	0	2
60	14	83	40
63	25	72	40
64	35	62	16
65	47	50	40
66	24	73	20
67	42	55	80
70	2	95	80
75	32	65	20
76	41	56	80
78	2	95	80
83	9	88	16
85	5	92	80
88	17	80	40
90	31	66	5
94	43	54	80
96	30	67	80

## ■ 4.2 Overview of Schoof's Algorithm

The method for determining  $\#E(\mathbb{F}_p)$  outlined above is feasible only for small  $p$ . In modern cryptographic applications of elliptic curves the cardinality of the field is typically a number with at least 50 decimal digits. Thus there is a need for an efficient means of computing  $\#E(\mathbb{F}_p)$  for large primes  $p$ . René Schoof's 1985 paper entitled "Elliptic curves over finite fields and the computation of square roots mod  $p$ ", details a polynomial time algorithm for determining  $\#E(\mathbb{F}_q)$ . Versions of this algorithm, enhanced by Elkies and Atkins, have been used successfully for  $q$  with hundreds of decimal digits [3,6]. The following steps sketch the outline of Schoof's method.

Let  $E$  be an elliptic curve over  $\mathbb{F}_q$  given by

$$E: y^2 = x^3 + Ax + B, \text{ where } A, B \in \mathbb{F}_q. \quad (13)$$

Hasse's Theorem tells us that the cardinality of the group of points is

$$\#E(\mathbb{F}_q) = q + 1 - t, \text{ with } |t| \leq 2\sqrt{q}. \quad (14)$$

Let  $\phi_q: E(\overline{\mathbb{F}_q}) \rightarrow E(\overline{\mathbb{F}_q})$  such that  $\phi_q((x, y)) = (x^q, y^q)$ . Note that this is map of points with coordinates in the algebraic closure of  $\mathbb{F}_q$ . Then  $\phi_q$  is an endomorphism called the Frobenius map. It has the following property, crucial to Schoof's algorithm.

$$\phi_q^2 - t\phi_q + q = 0 \quad \forall P \in E(\overline{\mathbb{F}_q}) \quad (15)$$

We can use (15) to compute  $t \pmod{p_i}$  for a set of  $L$  primes  $p_1, p_2, \dots, p_L$  such that

$$K = \prod_{i=1}^L p_i > 4\sqrt{q},$$

The Chinese Remainder Theorem is then applied to compute the unique

$$t \pmod{K} \text{ such that } |t| \leq 2\sqrt{q}.$$

Once we know  $t$  we can compute the order of the group as  $\#E(\mathbb{F}_q) = q + 1 - t$ . Schoof showed that this algorithm will run time proportional to  $\log^9 q$ , based on analysis of the number of elementary operations required. The following sec-



tions will explain the details of this algorithm, along with some important observations that permit its efficient implementation.

### ■ 4.3 Hasse's Theorem

The following theorem, first proved by Helmut Hasse in 1933, places specific bounds on  $\#E(\mathbb{F}_q)$ . Let  $E(\mathbb{F}_q)$  be an elliptic curve over the finite field  $\mathbb{F}_q$  with  $q = p^k$ ,  $k \in \mathbb{Z}^+$  and  $p$  a prime. Then there exists a unique  $t \in \mathbb{Z}$  such that

$$\#E(\mathbb{F}_q) = q + 1 - t \text{ where } |t| < 2\sqrt{q} \quad (16)$$

#### Sketch of the Proof

Define the map  $(\phi_q - 1): E(\overline{\mathbb{F}}_q) \rightarrow E(\overline{\mathbb{F}}_q)$ , then the set of points in  $E(\overline{\mathbb{F}}_q)$  which are sent to the identity by this map is called the kernel. Then  $\ker(\phi_q - 1) = E(\mathbb{F}_q)$ , since  $\phi_q$  is the identity on  $E(\mathbb{F}_q)$ . Further, since  $\phi_q - 1$  is a separable polynomial then

$$\#E(\mathbb{F}_q) = \# \ker(\phi_q - 1) = \deg(\phi_q - 1).$$

Now let  $t = q + 1 - \#E(\mathbb{F}_q)$ . Then by Washington [7] Proposition 3.16 for  $r, s \in \mathbb{Z}$  and  $\gcd(s, q) = 1$  we have  $\deg(r\phi_q - s)$

$$\begin{aligned} &= r^2(\deg \phi_q) + s^2 \deg(-1) + r s(\deg(\phi_q - 1) - \deg(\phi_q) - \deg(-1)) \\ &= r^2 q + s^2 + r s (\#E(\mathbb{F}_q) - q - 1) \\ &= r^2 q + s^2 + r s (q + 1 - t - q - 1). \end{aligned}$$

So we can conclude that

$$\deg(r\phi_q - s) = r^2 q + s^2 - r s t. \quad (17)$$

Since  $\deg(r\phi_q - s) \geq 0$  and  $s \neq 0$  then dividing through by  $s^2$  gives

$$q \left(\frac{r}{s}\right)^2 - \tau \left(\frac{r}{s}\right) + 1 \geq 0. \quad (18)$$

Having that the set of rational numbers  $\frac{r}{s}$  with  $\gcd(s, q) = 1$  is dense in  $\mathbb{R}$  implies that for all  $x \in \mathbb{R}$  we have

$$q x^2 - \tau x + 1 \geq 0. \quad (19)$$

So quadratic equation (19) has no real roots, hence its discriminant is less than zero. Thus

$$\tau^2 - 4q < 0 \Rightarrow |\tau| < 2\sqrt{q},$$

completing the proof of (16).

#### ■ 4.4 Reducing the problem to that for $E(\mathbb{F}_p)$

A beautiful result due to Andre Weil, and explained in Washington [7] Theorem 4.12, shows that if we can compute  $\#E(\mathbb{F}_p)$ , then we can compute  $\#E(\mathbb{F}_{p^n})$  in a direct manner.

Let  $\#E(\mathbb{F}_p) = p + 1 - t$ . Write  $X^2 - tX + p = (X - \alpha)(X - \beta)$ . Then  $\alpha^n + \beta^n \in \mathbb{Z}$  and

$$\#E(\mathbb{F}_{p^n}) = p^n + 1 - (\alpha^n + \beta^n). \quad (20)$$

So we only need to use Schoof's Algorithm to solve for  $\#E(\mathbb{F}_p)$ . Then we can compute  $\#E(\mathbb{F}_q) = \#E(\mathbb{F}_{p^n})$  via (20). Of course, if  $p$  is a small prime, then it is easy to determine  $\#E(\mathbb{F}_p)$  by direct counting or other simple methods, so the complexity of Schoof's method would not be justified. Assuming  $p$  is large enough to warrant the use of Schoof's method we may employ another useful result allowing us to determine the integer  $(\alpha^n + \beta^n)$  without explicitly computing  $\alpha$  and  $\beta$ . The following recursion relation computes  $s_n = (\alpha^n + \beta^n)$  where

$$s_0 = 2, \quad s_1 = t, \quad s_{n+1} = t s_n - p s_{n-1}. \quad (21)$$

The *Mathematica* function `ComputeOrderEFq[t, p, n]` implements equations (21) and (22).

For our example elliptic curve  $E: y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$  we determined that  $\#E(\mathbb{F}_{97}) = 80$ . By Hasse's theorem  $\#E(\mathbb{F}_{97}) = p + 1 - t$  so that  $80 = 97 + 1 - t$ , hence  $t = 18$ . Then we can determine  $\#E(\mathbb{F}_{97^4})$  using `ComputeOrderEFq`, giving  $\#E(\mathbb{F}_{p^4}) = 88531200$ .

#### ■ 4.5 Baby Step, Giant Step Method

One way to use Hasse's bound to compute  $\#E(\mathbb{F}_p)$  is based on Lagrange's theorem which states that the order of any element of a finite group must divide the order of the group. Hence if we can find the order  $k$  of a point  $Q \in E(\mathbb{F}_p)$  then the group order must be a multiple of  $k$  falling inside of Hasse's bounds. If we compute the order for several different points then some common multiple of these orders must fall inside of Hasse's bounds. Let  $\{k_i\}$  be the set of orders of  $n$  points in  $E(\mathbb{F}_p)$ . By Hasse's theorem we have for some integer  $r$  that  $p + 1 - 2\sqrt{p} < r * \text{lcm}(\{k_i\}) < p + 1 + 2\sqrt{p}$ . If there is only one  $r$  for which this is true, then we must have  $\#E(\mathbb{F}_p) = r * \text{lcm}(\{k_i\})$ .

In order for this method to be efficient we need a high performance method to compute point orders, that is, a method far better than exhaustive search to find the smallest  $k$  such that  $kP = O$  in  $E(\mathbb{F}_p)$ . The Baby Step-Giant Step method outlined in Washington [7] § 4.3 provides such a method with runtime proportional to  $\sqrt[4]{p}$ .

#### ■ Example 8 - Determining Group Order using Hasse's Theorem

For this example we again take  $E: y^2 = x^3 + Ax + B$  over  $\mathbb{F}_{97}$ . Since  $9 < \sqrt{97} < 10$ , Hasse's theorem gives

$$97 - 2 * 9 = 79 \leq \#E(\mathbb{F}_q) \leq 117 = 97 + 2 * 9.$$

We randomly found that  $P = (64, 35)$  is a point on the curve, and that  $|P| = 16$ . Since there are three multiples of 16 between 79 and 117, we need to choose another point. We randomly found a second point  $Q = (46, 95)$  with  $|Q| = 20$ . Then  $\text{lcm}(16, 20) = 80$ . Since  $79 \leq 80 \leq 117$  we can conclude that  $\#E(\mathbb{F}_q) = 80$ , in agreement with the direct counting method.

## Chapter 5 - Schoof's Algorithm Implementation

"A four-year-old child could understand that. Run out and find me a four-year-old child, I can't make head or tail out of it." - Groucho Marx (*Duck Soup-1933*)

In this chapter we present the algorithms that embody the key ideas of Schoof's method. For the curve given by  $E : y^2 = x^3 + Ax + B$ , over  $\mathbb{F}_p$ , Hasse's theorem tells us that  $\#E(\mathbb{F}_p) = p + 1 - t$ . The main objective of Schoof's algorithm is to determine  $t \pmod{l}$  for a set of small primes  $l$ . For the case of  $l = 2$  we have a special method, so we outline this first. For  $l > 2$  we must employ more sophisticated mathematics including the Frobenius endomorphism and the so-called division polynomials. We will examine these in more detail after describing the method for computing  $t \pmod{2}$ .

### ■ 5.1 Computing $t \pmod{2}$

As before, let  $E$  be an elliptic curve over the finite field  $\mathbb{F}_p$  given by  $E: y^2 = x^3 + Ax + B$ . A point  $P \in E(\mathbb{F}_p)$  has order 2 if and only if  $P + P = O$  which means that  $P = -P$ . As we have seen, this is true only if the  $y$ -coordinate of  $P$  is zero. Now  $y = 0$  if and only if  $x^3 + Ax + B = 0$ .

Suppose that there exists some  $e \in \mathbb{F}_p$  such that  $e^3 + Ae + B = 0$  then  $(e, 0) \in E(\mathbb{F}_p)$ . Also, by the definition of elliptic curve addition,  $2(e, 0) = \infty$ , so that  $(e, 0) \in E[2]$ . Then  $E(\mathbb{F}_p)$  has a point of order 2, so that, by Langrange's Theorem,  $\#E(\mathbb{F}_p) = p + 1 - t$  is even. Since  $p + 1$  is even then  $t$  also is even, therefore  $t \equiv 0 \pmod{2}$ .

Alternatively, suppose that  $\#E(\mathbb{F}_q)$  is even. Then by Theorem 4.1 of Washington [7] either

$$E(\mathbb{F}_p) \cong \mathbb{Z}_n \text{ or } E(\mathbb{F}_p) \cong \mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2} \text{ with } n, n_1, n_2 \in \mathbb{Z} \text{ and } n_1 \mid n_2.$$

If two groups are isomorphic, there is a 1-1 mapping between the elements of the two groups which preserves the group operation. This means, in particular, that if one has a nonzero point of order 2, then the other has a nonzero point of order 2.

If  $E(\mathbb{F}_p) \cong \mathbb{Z}_n$  then  $n$  is even, because  $\#E(\mathbb{F}_p)$  is even. We know that  $\mathbb{Z}_n$  is cyclic with generator 1, and  $2 * (\frac{n}{2} * 1) = n \equiv 0 \pmod{n}$ , so  $\frac{n}{2}$  is an element of  $\mathbb{Z}_n$  of order 2, therefore  $E(\mathbb{F}_p)$  is cyclic with some generator  $P_g$  and  $2 * (\frac{n}{2} * P_g) = n P_g = \infty$ , so that  $P_2 = \frac{n}{2} P_g$  has order 2, therefore  $P_2 = (e, 0)$  for some  $e \in \mathbb{F}_p$ .

Otherwise,  $E(\mathbb{F}_p) \cong \mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2}$  and  $n_1 * n_2$  is even, so that either  $n_2$  is even or  $n_1$  is even, which implies that  $n_2$  is even, because  $n_1 \mid n_2$ . So we have that

$$(0, \frac{n_2}{2}) \text{ is a point of order 2 in } \mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2},$$

therefore  $E(\mathbb{F}_p)$  has a point of order 2, call it  $P_2$ . So we conclude that  $\#E(\mathbb{F}_q) \equiv 0 \pmod{2} \Rightarrow \exists P_2 \in E(\mathbb{F}_p)$  with  $2P_2 = \infty$ .

The contrapositive gives that if  $E(\mathbb{F}_p)$  does not have a point of order 2, then

$\# E(\mathbb{F}_q) \equiv 1 \pmod{2} \Rightarrow t \equiv 1 \pmod{2}$ . Hence to compute  $t \pmod{2}$  it suffices to determine if  $x^3 + Ax + B$  has a root in  $\mathbb{F}_p$ .

### ■ 5.2 Determining if $x^3 + Ax + B$ has a root in $\mathbb{F}_q$

A basic theorem of algebra tells us if  $g(x)$  is a polynomial of degree  $n$  with coefficients in  $\mathbb{F}_p$  then  $g(x)$  has  $n$  roots in  $\overline{\mathbb{F}}_p$ , the algebraic closure of  $\mathbb{F}_p$ . In addition if  $g(x)$  has no roots in common with  $g'(x)$ , then the roots of  $g(x)$  are distinct.

Take  $g(x) = x^p - x$ . Then  $g'(x) = px^{p-1} - 1 = -1$ , since  $p \equiv 0 \pmod{p}$ , so  $g'(x)$  has no roots in  $\overline{\mathbb{F}}_p$ . Therefore, the  $p$  roots of  $g(X)$  are distinct, and these are just the set of elements  $\alpha$  of  $\overline{\mathbb{F}}_p$  satisfying  $\alpha^p - \alpha = 0 \Rightarrow \alpha^{p-1} = 1$ . But the  $p - 1$  non-zero elements of  $\mathbb{F}_p^\times$  all have order  $p - 1$ , so the roots of  $g(X)$  are precisely the elements of  $\mathbb{F}_p$ .

Then  $x^3 + Ax + B$  has a root in  $\mathbb{F}_p$  if and only if it has a root in common with  $g(x)$ . So if  $\gcd(x^3 + Ax + B, x^p - x) = 1$ , then  $x^3 + Ax + B$  has no root in  $\mathbb{F}_p$ , else it has at least one such root. From a practical standpoint we can compute  $x_p \equiv x^p \pmod{x^3 + Ax + B}$  using an efficient algorithm for modular polynomial exponentiation, and then compute

$$g = \gcd(x^3 + Ax + B, x_p - x) = \gcd(x^3 + Ax + B, x^p - x),$$

using the Euclidean algorithm for polynomials.

Given  $g$ , we determine  $t \pmod{2}$  as

$$t \equiv 1 \pmod{2} \text{ if } g = 1, \text{ else } t \equiv 0 \pmod{2}.$$

This method is encoded in the *Mathematica* function `ComputeTModTwo`.

■ **Example 9 - Computation of  $t \pmod{2}$**

We know from previous examples that for  $E : y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$  that  $\#E(\mathbb{F}_p) = 80$ . Hasse's theorem tells us that  $\#E(\mathbb{F}_p) = p + 1 - t$ , hence  $t = 18$ , so that  $t \equiv 0 \pmod{2}$ . By the previous discussion  $E(\mathbb{F}_p)$  has a point of order two if and only if

$$\gcd(x^p - x, x^3 + 46x + 74) \neq 1$$

where we can compute  $x^p - x$  modulo  $x^3 + 46x + 74$ . We find, using modular polynomial arithmetic, that

$$x^p \pmod{x^3 + 46x + 74} = 30x^2 + 60x + 47.$$

Then using a modular polynomial version of the Euclidean algorithm we compute

$$\gcd(30x^2 + 59x + 47, x^3 + 46x + 74) = x + 40 \neq 1.$$

Hence  $E(\mathbb{F}_p)$  has at least one point of order two. In fact, the table in Figure 3 shows it has exactly one such point, namely  $P = (57, 0)$ , thus  $\#E(\mathbb{F}_p)$  is even. Since  $\#E(\mathbb{F}_p) = p + 1 - t$ , and  $p + 1 = 98$  is even, then  $t$  is even. Hence  $t \equiv 0 \pmod{2}$ .

### ■ 5.3 The Division Polynomials

In order to determine  $t \pmod{l_i}$  for primes  $l_i > 2$ , we need to make use of what are called the *division polynomials* for  $E : y^2 = x^3 + ax + b$ . These are polynomials which go to zero on points of a particular order. We define  $E[n]$  as the set of  $n$ -torsion points of an elliptic curve  $E : y^2 = x^3 + ax + b$ , that is, the set of points in  $E(\overline{\mathbb{F}}_p)$  with order dividing  $n$ , so that  $E[n] = \{P \in E(\overline{\mathbb{F}}_p) \mid nP = O\}$ . Note that this set includes points with coordinates in  $\overline{\mathbb{F}}_p$ , the algebraic closure of  $\mathbb{F}_p$ .

With this definition the division polynomials  $\psi_n$  of an elliptic curve  $E$  are elements of  $\mathbb{F}_p[x, y]$  with the property that  $\psi_n(x, y) = 0$  if and only if  $(x, y) \in E[n]$ . These polynomials are defined recursively as follows.

$$\begin{aligned}\psi_0 &= 0, \psi_1 = 1, \psi_2 = 2y, \\ \psi_3 &= 3x^4 + 6ax^2 + 12bx - a^2 \\ \psi_4 &= 4y(x^6 + 5ax^4 + 20bx^3 - 5a^2x^2 - 4abx - 8b^2 - a^3) \\ \psi_{2n} &= \psi_n(\psi_{n+2}\psi_{n-1}^2 - \psi_{n-2}\psi_{n+1}^2) & n \in \mathbb{Z}, n > 2 \\ \psi_{2n+1} &= \psi_{n+2}\psi_n^3 - \psi_{n+1}^3\psi_{n-1} & n \in \mathbb{Z}, n > 1\end{aligned}$$

Lets see why  $\psi_3$  is the correct polynomial. First, if  $P = (x, y) \in E[3]$  then  $3P = O$  which means that  $2P = -P$ , hence the  $x$ -coordinates of  $2P$  and  $-P$  must be the same. Using Equations (11,12) to compute  $2P$  we find

$$x = \lambda^2 - 2x = \frac{(3x^2+A)^2}{4y^2} - 2x$$

so that

$$(-3x)(4y^2) = 9x^4 + 6Ax^2 + A^2.$$

But  $y^2 = x^3 + Ax + B$  so that  $-12(x^4 + Ax^2 + Bx) = 9x^4 + 6Ax^2 + A^2$ . Collecting terms and multiplying through by  $-1$  gives

$$3x^4 + 6Ax^2 + 12Bx - A^2 = \psi_3.$$

So if  $\psi_3 = 0$  then  $2P = \pm P$ , meaning that  $P = O$  or  $P$  is a point of order 3. In either case  $P \in E[3]$ .

The division polynomials are polynomials in  $x, y$ . Using the elliptic curve equation we can replace  $y^2$  with  $x^3 + Ax + B$ . More generally we can replace  $y^{2k}$



with  $(x^3 + Ax + B)^k$ . This allows us to express the division polynomials as elements of  $\mathbb{F}_p[x]$  or  $y\mathbb{F}_p[x]$ , so that no power of  $y$  greater than 1 will appear. It can be further proved that we can produce polynomials in  $\mathbb{F}_p[x]$  with the following replacements.

$$f_n(x) = \begin{cases} \psi_n(x, y) & \text{if } n \text{ is odd} \\ \psi_n(x, y)/y & \text{if } n \text{ is even} \end{cases}$$

These polynomials, by definition, also have the property that  $f_n(x) = 0$  if and only if  $x$  is the  $x$ -coordinate of a point of order  $n$ .

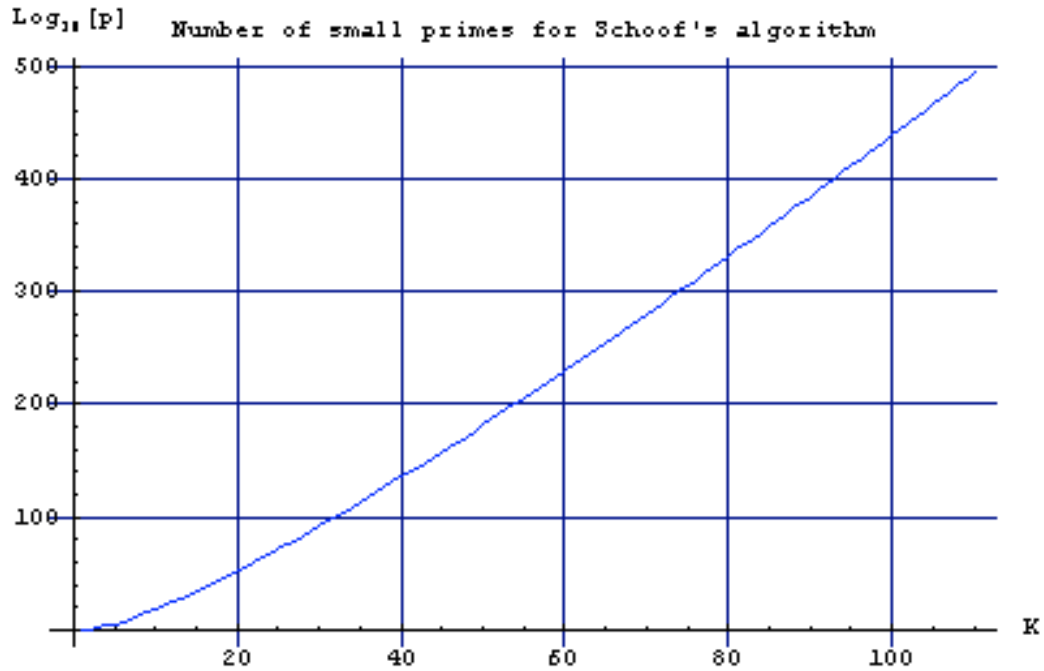
#### ■ 5.4 How many division polynomials?

How many division polynomials will we need for the execution of Schoof's algorithm? As noted in the outline of Schoof's algorithm in chapter 4, we need to test Equation (15) for a set of primes  $l_i$  such the product of these primes is greater than  $4\sqrt{p}$ . The function `ComputePrimeSet` determines this set. If the cardinality of  $\{l_i\} = k$ , then  $\psi_{k+2}$  is the highest order division polynomial required.

What is the relationship between  $k$  and  $p$ ? Figure 4 contains a plot of  $\text{Log}_{10}[p]$  vs.  $k$ , which indicates that  $k$  grows approximately logarithmically with  $p$ . The horizontal axis is the number of primes  $k$ , the vertical axis is the number of decimal digits in  $p$ , the size  $\mathbb{F}_p$ . A statistical fit of this data gives the approximate relationship for  $k > 10$ .

$$\text{Log}_{10}[p] = 0.012 k^2 + 3.34 k - 15.98$$

Given that we wish to apply Schoof's algorithm to an elliptic curve over  $\mathbb{F}_p$  we could use this graph to estimate the number of small primes  $k$  that would be required.

Figure 3 - Number of digits in  $p$  vs. number of small primes.

■ Example 10 - Computation of the Division Polynomials

For our example  $E : y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$ , we first compute the set of small primes whose product is greater than  $4\sqrt{97}$ , such that  $p \not\equiv 1 \pmod{l_i}$  for  $l_i > 2$ . The necessary primes are 2, 5, 7 whose product is  $70 > 4\sqrt{97}$ . Then  $p \equiv 2 \pmod{5}$ , and  $p \equiv 6 \pmod{7}$ . Therefore we will need division polynomials up to and including  $\psi_9$ , so we compute these at this time. Note that the odd numbered polynomials, such as  $\psi_1, \psi_3, \dots$  are polynomials in  $x$  only, while the even numbered polynomials are polynomials in  $x$  multiplied by  $y$ . More precisely  $\psi_{2n+1} \in \mathbb{F}_p[x]$  and  $\psi_{2n} \in y\mathbb{F}_p[x]$ .

For our sample curve we find that the first five division polynomials are

$$\psi_1(x, y) = 1,$$

$$\psi_2(x, y) = 2y,$$

$$\psi_3(x, y) = 18 + 15x + 82x^2 + 3x^4,$$

$$\psi_4(x, y) = (61 + 50x + 69x^2 + 3x^3 + 47x^4 + 4x^6)y,$$

$$\begin{aligned} \psi_5(x, y) = & 23 + 67x + 11x^2 + 38x^3 + 77x^4 + 43x^5 + 93x^6 \\ & + 26x^7 + 47x^8 + 87x^9 + 39x^{10} + 5x^{12}. \end{aligned}$$

With  $E : y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$  we have that  $(4, 15)$  is a point of order 4. Then we must have  $f_n[4] = 0$  if and only if  $4 \mid n$ . To check the function **ComputeDivisionPolynomials** we calculate  $f_n[4]$  for  $2 \leq n \leq 8$  giving

$$\begin{aligned} f_2[4] &= 2, f_3[4] = 24, f_4[4] = 0, f_5[4] = 47, \\ f_6[4] &= 25, f_7[4] = 22, f_8[4] = 0, \end{aligned}$$

as expected, since  $(4, 15) \in E[4]$  and  $E[4] \subseteq E[8]$ . Similarly the point  $(90, 31)$  is of order 5 and we find

$$f_2[90] = 2, f_3[90] = 76, f_4[90] = 14, f_5[90] = 0, f_6[90] = 21, f_7[90] = 23.$$

So the division polynomials are correct, at least for this particular case.

### ■ 5.5 Computing $nP$ with the Division Polynomials

If  $P = (x, y)$  is a point in  $E(\overline{\mathbb{F}}_p)$  then

$$nP = \left( x - \frac{\psi_{n-1} \psi_{n+1}}{\psi_n^2}, \frac{\psi_{n+2} \psi_{n-1}^2 - \psi_{n-2} \psi_{n+1}^2}{4y \psi_n^3} \right) \quad (22)$$

It can be shown that multiplication by  $n$  is an endomorphism  $\mu_n$  of  $E(\overline{\mathbb{F}}_p)$ . This follows from the fact that  $E(\overline{\mathbb{F}}_p)$  is an abelian group so that  $n(P + Q) = nP + nQ$ . Since  $nP = O$  if and only if  $P \in E[n]$  we have that the kernel of  $\mu_n$  is  $E[n]$ . Further, because  $\mu_n$  is expressed as a separable rational polynomial of degree  $n^2$ , we have that  $\# E[n] = \deg(\mu_n) = n^2$ . For a proof of (21) see Washington [7] § 9.5.

It should be noted that Equation (21) does not provide an efficient way to compute  $nP$  for specific points  $P$ . Rather, it provides the basis of proof for the characteristic equation of the Frobenius (15), one of the key equations used in Schoof's method.

### ■ 5.6 The Frobenius Endomorphism

Let  $\phi_q : E(\overline{\mathbb{F}}_q) \rightarrow E(\overline{\mathbb{F}}_q)$  with  $\phi_q(x, y) = (x^q, y^q)$ , called the Frobenius endomorphism. Since  $a^q = a$  for all  $a \in \mathbb{F}_q$  this map is the identity for points with coordinates in  $\mathbb{F}_q$ . Let  $(x_1, y_1) \in E(\overline{\mathbb{F}}_q)$  then

$$y_1^2 = x_1^3 + A x_1 + B \text{ in } \overline{\mathbb{F}}_q.$$

Now  $\phi_q(x_1, y_1) = (x_1^q, y_1^q)$ . Substituting this into the elliptic curve equation gives

$$(y_1^q)^2 = (y_1^2)^q = (x_1^3 + A x_1 + B)^q.$$

However, for all  $a, b \in \overline{\mathbb{F}}_q$  we have  $(a + b)^q = a^q + b^q$  in  $\mathbb{F}_q$  so that

$$(y_1^q)^2 = (x_1^3)^q + A^q x_1^q + (B)^q = (x_1^q)^3 + A x_1^q + B, \text{ since } A, B \in \mathbb{F}_q.$$

Hence  $(x_1^q, y_1^q) = \phi_q(x_1, y_1) \in E(\overline{\mathbb{F}}_q)$ , so that  $\phi_q$  maps a point on the curve to another point on the curve.

Let  $P = (x_1, y_1)$ ,  $Q = (x_2, y_2)$  be two points in  $E(\overline{\mathbb{F}}_q)$  with  $x_1 \neq x_2$ , then with  $\lambda = (y_2 - y_1)/(x_2 - x_1)$  we have

$$P + Q = (x_3, y_3) \text{ with } x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1.$$

Using the same properties of  $q^{\text{th}}$  powers in  $\overline{\mathbb{F}}_q$  we have

$$\phi_q(P + Q) = (\lambda^{2q} - x_1^q - x_2^q, \lambda^q(x_1^q - x_3^q) - y_1^q),$$

$$\text{with } \lambda^q = (y_2^q - y_1^q)/(x_2^q - x_1^q).$$

Therefore  $\phi_q(P + Q) = \phi_q(P) + \phi_q(Q)$ . It can also be shown that this holds also for  $Q = P$  and  $Q = -P$ , so that  $\phi_q$  is a homomorphism from  $E(\overline{\mathbb{F}}_q)$  to  $E(\overline{\mathbb{F}}_q)$ , hence an endomorphism.

### ■ 5.7 The Characteristic Equation of the Frobenius

We now proceed to the equation that provides the foundation for Schoof's algorithm. Remember that  $E[l]$  is the set of points in  $E(\overline{\mathbb{F}}_q)$  whose order divides  $l$ . First we show that  $E[l]$  is a subgroup of  $E(\overline{\mathbb{F}}_q)$ . Clearly  $O \in E[l]$ . If  $P, Q \in E[l]$  then, because  $E(\overline{\mathbb{F}}_q)$  is abelian,  $O = lP + lQ = l(P + Q)$ , so that  $P + Q \in E[l]$ . Further,

$$O = lO = l(P + (-P)) = lP + l(-P) = O + l(-P) = l(-P),$$

therefore  $-P \in E[l]$ . Hence  $E[l]$  is a subgroup of  $E(\overline{\mathbb{F}}_q)$ .

Since  $E[l]$  is an abelian group of order  $l^2$ , it follows from the structure theorem for finite abelian groups that  $E[l] \cong \mathbb{Z}_l \oplus \mathbb{Z}_l$ . The integer  $l$  is a prime so that  $\mathbb{Z}_l$  is a cyclic group generated by any number  $1 \leq a < l$ , and there exists points  $\beta_1, \beta_2 \in E[l]$  such that any point in  $E[l]$  can be written as a  $\mathbb{Z}_l$ -linear combination  $P = m_1 \beta_1 + m_2 \beta_2$  with  $m_1, m_2 \in \mathbb{Z}_l$ . Suppose  $\alpha$  is any homomorphism of  $E[l]$ , then  $\alpha(P) \in E[l]$  for all  $P \in E[l]$  because  $|\alpha(P)|$  divides  $|P|$ . Then, in particular,  $\alpha(\beta_1) = s \beta_1 + t \beta_2$  and  $\alpha(\beta_2) = u \beta_1 + v \beta_2$ , so that

$$\begin{aligned} \alpha(P) &= \alpha(m_1 \beta_1 + m_2 \beta_2) = m_1 \alpha(\beta_1) + m_2 \alpha(\beta_2) \\ &= m_1 s \beta_1 + m_1 t \beta_2 + m_2 u \beta_1 + m_2 v \beta_2. \end{aligned}$$

We can express this in matrix form as

$$\alpha(P) = \begin{pmatrix} s & t \\ u & v \end{pmatrix} \begin{pmatrix} m_1 \beta_1 \\ m_2 \beta_2 \end{pmatrix}. \quad (23)$$

In particular, the action of the Frobenius on  $E[l]$ , denoted  $\phi_{q,l}$  can be described by such a  $2 \times 2$  matrix. Applied to  $E[l]$  we have

$$\begin{aligned} \deg(\phi_q - 1) &\equiv \det(\phi_{q,l} - I) \pmod{l} \\ &= (s - 1)(v - 1) - tu = sv - tu - (s + v) + 1. \end{aligned}$$

But  $sv - tu = \det(\phi_{q,l}) \equiv q \pmod{l}$  (by Washington [7] Proposition 3.15).

Then by Hasse's theorem, using  $a$  instead of  $t$  to avoid conflicting variable names, we have

$$\# \ker(\phi_q - 1) = q + 1 - a \equiv q - (s + v) + 1 \pmod{l}.$$

Hence we have the following congruences. First,  $a \equiv (s + v) \pmod{l}$ , the trace of  $\phi_{q,l}$ . Also  $tu \equiv sv - q \pmod{l}$ .

We can now compute  $(\phi_{q,l})^2 - a(\phi_{q,l}) + q$  using \*, so that

$$\begin{aligned} \begin{pmatrix} s & t \\ u & v \end{pmatrix}^2 - a \begin{pmatrix} s & t \\ u & v \end{pmatrix} + q \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} &= \begin{pmatrix} s^2 + tu & st + tv \\ su + uv & tu + v^2 \end{pmatrix} - \begin{pmatrix} as - q & at \\ ay & av - q \end{pmatrix} \\ &= \begin{pmatrix} s^2 + tu - as + q & st + tv - at \\ su + uv - au & v^2 + tu - av + q \end{pmatrix}. \end{aligned}$$

Applying the congruences yeild

$$\begin{aligned} &\equiv \begin{pmatrix} s^2 + sv - q - (s+v)s + q & (s+v)t - (s+v)t \\ (s+v)u - (s+v)u & v^2 + sv - q - (s+v)v + q \end{pmatrix} \\ &\equiv \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \pmod{l}. \end{aligned}$$

Therefore  $(\phi_{q,l})^2 - a(\phi_{q,l}) + q \equiv 0 \pmod{l}$  for all  $l$  such that  $\gcd(l, q) = 1$ . Since there are an infinite number of choices for such  $l$ , the kernel of  $\phi_q^2 + q - a\phi_q$  is not finite, hence, as stated in Equation (15),

$$\phi_q^2 + q - a\phi_q = 0 \text{ for all } P \in E(\overline{\mathbb{F}}_q).$$

In particular, if  $P \in E[l]$  then,

$$\phi_q^2 + k \equiv \tau \phi_q \pmod{l} \text{ where } k \equiv q \pmod{l} \text{ and } \tau \equiv a \pmod{l}.$$

So for a particular point  $P = (x, y) \in E[l]$  it must be true that

$$(x^{q^2}, y^{q^2}) + k(x, y) \equiv \tau(x^q, y^q) \pmod{l}, \tag{24}$$

where addition is performed in  $E(\overline{\mathbb{F}}_q)$  using (9) through (12), and scalar point multiplication is performed using the division polynomials as in Equation (22). We can simplify the computation of (24) further by noting that if  $(x, y) \in E[l]$

then the division polynomial  $\psi_l(x, y) = 0$ , so we can reduce (24) mod  $\psi_l$  without changing the set of points which satisfy the equation.

### ■ 5.8 Schoof's Algorithm: Case One

In order to use Equation (24) we must first test if  $\phi_l^2 P = \pm k P$  for some  $P \in E[l]$ . We can determine this by computing the test condition for the  $x$  coordinate using **Schoof (16)**:

$$x^{p^2} \equiv x - \frac{\psi_{k-1} \psi_{k+1}}{\psi_k^2} \pmod{f_l, p}.$$

This is true if and only if

$$p_{16}(x, y) = (x^{p^2} - x) \psi_k^2 - \psi_{k-1} \psi_{k+1} \equiv 0 \pmod{f_l, p}.$$

For  $k$  even,  $\psi_k = y f_k$  and since  $y^2 = x^3 + a x + b$  we obtain

$$p_{16}(x) = (x^{p^2} - x) f_k^2(x) (x^3 + a x + b) + f_{k-1}(x) f_{k+1}(x).$$

For  $k$  odd,  $\psi_{k-1} = y f_{k-1}$  and  $\psi_{k+1} = y f_{k+1}$  so that

$$p_{16}(x) = (x^{p^2} - x) f_k^2(x) + f_{k-1}(x) f_{k+1}(x) (x^3 + a x + b).$$

Notice that  $p_{16}(x, y)$  is a polynomial in  $x$  only. Hence if  $\gcd(p_{16}(x), f_l(x)) \neq 1$  then some point  $P$  exists in  $E[l]$  which satisfies  $\phi_l^2 P = \pm k P$ , so we are in case 1. Otherwise we must proceed to case 2 where we test equation (24) for various values of  $\tau$ .

### ■ 5.9 Schoof Equation (17)

Given that  $\phi_l^2 P = \pm k P$  for some  $P \in E[l]$ , then  $t \in \{0, -2w, 2w\}$  where  $w^2 \equiv k \pmod{l}$ . This is shown as follows. Suppose  $\phi_l^2 P = k P$ , then, by equation (24) we have  $2k \equiv \tau \phi_l$ . Squaring both sides gives  $4k^2 = t^2 \phi_l^2 = t^2 k$ , so that  $4k = t^2$ , then we must have that  $k$  is a quadratic residue. If so, find  $w$  such that  $w^2 \equiv k \pmod{l}$ , then  $4w^2 = t^2$  so that  $t = \pm 2w$ . Now we can compute

$$(\phi_l - w)(\phi_l + w) = \phi_l^2 - k = 0, \text{ so } \phi_l P = \pm w P.$$

If  $k$  is not a quadratic residue, we can not be in this case, hence  $\phi_l^2 P = -k P$  so that  $\phi_l^2 P + k P = 0 = \tau \phi_l P$  for all  $P \in E[l]$  so that  $\tau \equiv 0 \pmod{l}$ .

We can test if  $\phi_l P = \pm w P$  using the point multiplication formula (22) again yielding for the  $x$ -coordinate the test

$$x^P \equiv x - \frac{\psi_{w-1} \psi_{w+1}}{\psi_w^2} \pmod{f_l, p}.$$

Multiplying through by  $\psi_w^2$  produces Schoof equation (17)

$$p_{17}(x, y) = (x^P - x) \psi_w^2 - \psi_{w-1} \psi_{w+1}.$$

For  $w$  even or odd this can be reduced to a polynomial in  $x$  only, as in

$$p_{17}(x) = (x^P - x) f_w^2(x) (x^3 + a x + b) + f_{w-1}(x) f_{w+1}(x) \text{ } w \text{ even,}$$

$$p_{17}(x) = (x^P - x) f_w^2(x) + f_{w-1}(x) f_{w+1}(x) (x^3 + a x + b) \text{ } w \text{ odd.}$$

If  $\gcd(p_{17}(x), f_l(x)) = 1$  then we must have  $\phi_l^2 P = -k P$  so that  $t \equiv 0 \pmod{l}$ . Otherwise  $t \equiv \pm w \pmod{l}$  and we test the  $y$ -coordinate of  $\phi_l P = \pm w P$  to determine the sign.



### ■ 5.10 Schoof Equation (18)

After we know that  $\phi_l P = \pm w P$  we need test the  $y$ -coordinate of  $\phi_l P = w P$ . Equation (22) gives for the  $y$ -coordinate,

$$y^p \equiv \frac{\psi_{w+2} \psi_{w-1}^2 - \psi_{w-2} \psi_{w+1}^2}{4 y \psi_w^3} \pmod{\psi_l, p}.$$

Multiplying through by the denominator of the right hand side and collecting terms gives

$$p_{18}(x, y) = 4 \psi_w^3 y^{p+1} - \psi_{w+2} \psi_{w-1}^2 - \psi_{w-2} \psi_{w+1}^2.$$

For  $w$  even:

$$p_{18}(x) = 4 (y^2)^{(p+3)/2} f_w^3(x) - f_{w+2}(x) f_{w-1}^2(x) + f_{w-2}(x) f_{w+1}^2(x).$$

For  $w$  odd:

$$p_{18}(x) = 4 (y^2)^{(p-1)/2} f_w^3(x) - f_{w+2}(x) f_{w-1}^2(x) + f_{w-2}(x) f_{w+1}^2(x).$$

Notice that  $p_{18}(x)$  is also a polynomial in  $x$  only since all exponents of  $y$  are even. If  $\gcd(p_{18}(x), f_l(x)) = 1$  then there is no  $P \in E[l]$  for which  $\phi_l P = w P$ , so  $t \equiv -2w \pmod{l}$ , else such a point exists and  $t \equiv 2w \pmod{l}$ . This completes the equations required to test for case 1.

### ■ 5.11 Schoof's Algorithm: Case Two

If there is no  $P \in E[l]$  such that  $\phi_l^2 P = \pm k P$  then we are in case 2 so we need to test for each  $\tau \in \mathbb{Z}/l\mathbb{Z}^x$  if there exist  $P \in E[l]$  such that (24) holds. In order to perform this test we apply addition formulas (5), (9) and (10) to compute polynomials representing

$$(x^{p^2}, y^{p^2}) + k(x, y),$$

where  $k(x, y)$  is computed using the division polynomials and equation (22). Since we are in case 2 we know that  $x_1 \neq x_2$  so we can compute  $\lambda = (y_2 - y_1)/(x_2 - x_1)$ . We find

$$y_2 - y_1 = \frac{\psi_{k+2} \psi_{k-1}^2 - \psi_{k-2} \psi_{k+1}^2}{4 y \psi_k^3} - y^{p^2},$$

$$x_2 - x_1 = x - \frac{\psi_{k-1} \psi_{k+1}}{\psi_k^2} - x^{p^2}$$

So then

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} = \frac{(\psi_{k+2} \psi_{k-1}^2 - \psi_{k-2} \psi_{k+1}^2 - 4 \psi_k^3 y^{p^2+1})}{4 y \psi_k (-\psi_{k-1} \psi_{k+1} - \psi_k^2 (x^{p^2} - x))}.$$

Put  $\lambda = \alpha / \beta$  then

$$\alpha = \psi_{k+2} \psi_{k-1}^2 - \psi_{k-2} \psi_{k+1}^2 - 4 \psi_k^3 y^{p^2+1}$$

and

$$\beta = 4 y \psi_k (\psi_k^2 (x - x^{p^2}) - \psi_{k-1} \psi_{k+1}).$$

For  $k$  even we have

$$\alpha = y(f_{k+2} f_{k-1}^2 - f_{k-2} f_{k+1}^2 - 4 f_k^3 y^{p^2+3})$$

and

$$\beta = 4 y^2 f_k (y^2 f_k^2 (x - x^{p^2}) - f_{k-1} f_{k+1}).$$

Otherwise, for  $k$  odd

$$\alpha = y^2 (f_{k+2} f_{k-1}^2 - f_{k-2} f_{k+1}^2) - 4 f_k^3 y^{p^2+1}$$

and

$$\beta = y(4 f_k (f_k^2 (x - x^{p^2}) - y^2 f_{k-1} f_{k+1})).$$

We use these equations for  $\alpha$  and  $\beta$  to formulate the tests of Schoof equations (19).

### ■ 5.12 Schoof Equation (19x)

Using the equations we just derived for  $\lambda = \alpha / \beta$  we can now compute the addition of points using equation (22) so that if

$$(x_3, y_3) = (x^{p^2}, y^{p^2}) + k(x, y)$$

then  $x_3$  is given by

$$x_3 = \lambda^2 - x_1 - x_2 = \frac{\alpha^2}{\beta^2} - (x^{p^2} + x) + \frac{\psi_{k-1} \psi_{k+1}}{\psi_k^2}.$$

Also  $y_3$  is given by

$$y_3 = \lambda(2x_1 + x_2 - \lambda^2) - y_1 = \frac{\alpha}{\beta} \left( 2x^{p^2} + x - \frac{\psi_{k-1} \psi_{k+1}}{\psi_k^2} - \frac{\alpha^2}{\beta^2} \right) - y^{p^2}.$$

Further, since  $\psi_n(x^p, y^p) = \psi_n(x, y)^p$  we have

$$\tau(x^p, y^p) = \left( \left( x - \frac{\psi_{\tau-1} \psi_{\tau+1}}{\psi_\tau^2} \right)^p, \left( \frac{\psi_{\tau+2} \psi_{\tau-1}^2 - \psi_{\tau-2} \psi_{\tau+1}^2}{4y \psi_\tau^3} \right)^p \right).$$

Then (23) holds if and only if (for the  $x$ -coordinate)

$$\frac{\alpha^2}{\beta^2} - (x^{p^2} + x) + \frac{\psi_{k-1} \psi_{k+1}}{\psi_k^2} = \left( x - \frac{\psi_{\tau-1} \psi_{\tau+1}}{\psi_\tau^2} \right)^p.$$

Expanding and clearing the (nonzero) denominators gives

$$\begin{aligned} & \psi_\tau^{2p} (\alpha \psi_k^2 - \beta^2 \psi_k^2 (x^{p^2} + x) + \beta^2 (\psi_{k-1} \psi_{k+1})). \\ & = \psi_k^2 \beta^2 (\psi_\tau^{2p} x^p - (\psi_{\tau-1} \psi_{\tau+1})^p) \end{aligned}$$

Bringing everything to the left hand side we have

$$\begin{aligned} p_{19_x}(x, y) &= \psi_\tau^{2p} (\beta^2 (\psi_{k-1} \psi_{k+1} - \psi_k^2 (x^{p^2} + x) + \alpha \psi_k^2)), \\ &+ \psi_k^2 \beta^2 (\psi_{\tau-1} \psi_{\tau+1})^p = 0 \end{aligned}$$

which is Schoof equation (19) for the  $x$ -coordinate. Since we are testing this equation for points in  $E[l]$  we perform all of the polynomial arithmetic modulo  $\psi_l$ .

Now there exists a point in  $E[l]$  satisfying  $p_{19_x}$  if and only if  $\gcd(p_{19_x}, f_l) \neq 1$ . If such a point exist, then  $t \equiv \pm \tau \pmod{l}$ . We use Schoof equation (19y), explained in the next section, to determine the sign of  $\tau$ .

### ■ 5.13 Schoof Equation (19y)

Once we know that there exists  $(x, y) \in E[l]$  such that

$$(x^{p^2}, y^{p^2}) + k(x, y) = \pm \tau(x^p, y^p),$$

we must test the  $y$ -coordinate to determine the sign of  $\tau$ . We have from the previous section,

$$y_3 = \frac{\alpha}{\beta} \left( 2 x^{p^2} + x - \frac{\psi_{k-1} \psi_{k+1}}{\psi_k^2} - \frac{\alpha^2}{\beta^2} \right) - y^{p^2}.$$

Then if (24) holds for the  $y$ -coordinate we have

$$\frac{\alpha}{\beta} \left( 2 x^{p^2} + x - \frac{\psi_{k-1} \psi_{k+1}}{\psi_k^2} - \frac{\alpha^2}{\beta^2} \right) - y^{p^2} = \left( \frac{\psi_{t+2} \psi_{t-1}^2 - \psi_{t-2} \psi_{t+1}^2}{4 y \psi_t^3} \right)^p.$$

Multiplying through by  $\beta^3 \psi_k^2 4 y^p \psi_t^3$  to clear the denominators we have

$$\begin{aligned} 4 y^p \psi_t^3 \left( \alpha \beta^2 (\psi_k^2 (2 x^{p^2} + x) - \psi_{k-1} \psi_{k+1}) - \psi_k^2 (\alpha^3 + \beta^3 y^{p^2}) \right) \\ = \beta^3 \psi_k^2 (\psi_{t+2} \psi_{t-1}^2 - \psi_{t-2} \psi_{t+1}^2)^p. \end{aligned}$$

Rearranging gives for Schoof (19y) (corrected)

$$\begin{aligned} p_{19_y}(x, y) = 4 f_\tau^3 y^p \left( ((2 x^{p^2} + x) \alpha \beta^2 - \beta^3 y^{p^2} - \alpha^3) f_k^2 - \alpha \beta^2 f_{k-1} f_{k+1} \right) \\ - \beta^3 f_k^2 (f_{\tau-1}^2 f_{\tau+2} - f_{\tau-2} f_{\tau+1}^2)^p. \end{aligned}$$

For  $k$  even,  $\tau$  even, we must take  $\alpha \rightarrow y \alpha$  so that,

$$\begin{aligned} p_{19_y}(x) = 4 f_\tau^3 y^3 y^{p-1} \left( ((2 x^{p^2} + x) \alpha \beta^2 - \beta^3 y^{p^2-1} - y^2 \alpha^3) y^2 f_k^2 \right. \\ \left. - \alpha \beta^2 f_{k-1} f_{k+1} \right) - \beta^3 f_k^2 (f_{\tau-1}^2 f_{\tau+2} - f_{\tau-2} f_{\tau+1}^2)^p. \end{aligned}$$

For  $k$  even,  $\tau$  odd,

$$p_{19,y}(x) = 4 f_{\tau}^{3p} (((2x^{p^2} + x) \alpha \beta^2 - \beta^3 y^{p^2-1} - \alpha^3 y^2) y^2 f_k^2 - \alpha \beta^2 f_{k-1} f_{k+1}) - \beta^3 f_k^2 y^{p+1} (f_{\tau-1}^2 f_{\tau+2} - f_{\tau-2} f_{\tau+1})^p.$$

For  $k$  odd,  $\tau$  even we must take  $\beta \rightarrow y\beta$  so that,

$$p_{19,y}(x) = 4 f_{\tau}^{3p} y^{3p-3} (((2x^{p^2} + x) \alpha y^2 \beta^2 - \beta^3 y^{p^2+3} - \alpha^3) f_k^2 - \alpha y^2 \beta^2 f_{k-1} f_{k+1}) - \beta^3 f_k^2 (f_{\tau-1}^2 f_{\tau+2} - f_{\tau-2} f_{\tau+1})^p.$$

For  $k$  odd,  $\tau$  odd,

$$p_{19,y}(x) = 4 f_{\tau}^{3p} (((2x^{p^2} + x) \alpha y^2 \beta^2 - \beta^3 y^{p^2+3} - \alpha^3) f_k^2 - \alpha y^2 \beta^2 f_{k-1} f_{k+1}) - \beta^3 f_k^2 y^{p+3} (f_{\tau-1}^2 f_{\tau+2} - f_{\tau-2} f_{\tau+1})^p.$$

Now if  $\gcd(p_{19,y}, f_l) \neq 1$  then  $E[l]$  has a point satisfying  $\phi_l^2 P + kP = \tau\phi P$ , so that  $t \equiv \tau \pmod{l}$ , else  $t = -\tau \pmod{l}$ .

### ■ 5.14 Schoof's Algorithm Summary

We can now summarize Schoof's algorithm for  $E : y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$  as follows. By Hasse's theorem we have  $\# E(\mathbb{F}_p) = p + 1 - t$ .

1. If  $\gcd(x^3 + ax + b, x^p - x) = 1$  then  $t \equiv 0 \pmod{2}$ , else  $t \equiv 1 \pmod{2}$
2. Create a set of small primes  $S = \{l_i\}$  such that  $\prod_{i=1}^L l_i > 4\sqrt{p}$ .
3. Compute the first  $l_L + 2$  division polynomials  $\psi_k$ .
4. For each  $l \in S$ , compute  $k \equiv p \pmod{l}$
5. If  $\gcd(p_{16}, f_l) \neq 1$  then there exists  $P \in E[l]$  such that  $\phi_l^2 P = \pm k P$ .
6. If  $k$  is not a quadratic residue mod  $l$ , then  $t \equiv 0 \pmod{l}$  else
7. Compute  $w$  such that  $w^2 \equiv k \pmod{l}$
8. If  $\gcd(p_{17}, f_l) = 1$  then  $t \equiv 0 \pmod{l}$ , else
9. If  $\gcd(p_{18}, f_l) \neq 1$  then  $t \equiv 2w \pmod{l}$ , else  $t \equiv -2w \pmod{l}$ .
10. else we are in case two
11. For each  $\tau \leq (l+1)/2$
12. If  $\gcd(p_{19}, f_l) \neq 1$  then  $\phi_p^2 + k \equiv \pm \tau \phi_p \pmod{l}$
13. for some point in  $E[l]$  so we test
14. If  $\gcd(p_{19}, f_l) \neq 1$  then  $t \equiv \tau \pmod{l}$  else  $t \equiv -\tau \pmod{l}$
15. Next  $\tau$
16. Next  $l$
17. At this point we have computed  $t \pmod{l_i}$  for all  $l_i \in S$ ,
18. so we use the Chinese Remainder Theorem to compute
19.  $T \equiv t \pmod{N}$  where  $N = \prod_{i=1}^L l_i$ .
20. If  $T$  is within Hasse's bounds then  $t = T$ , else  $t \equiv -T \pmod{N}$  and
21.  $\# E(\mathbb{F}_p) = p + 1 - t$ .

This completes the description of Schoof's algorithm.

## Chapter 6 - Results of Running Schoof's Algorithm

This chapter contains the results of running our implementation of Schoof's algorithm for several different elliptic curves. We present detailed results for one particular curve and then summarize the results for other curves in table 1. We conclude this section with a discussion of lessons learned from these experiments.

### ■ 6.1 A Detailed Example

For our example curve  $E : y^2 = x^3 + 46x + 74$  over  $\mathbb{F}_{97}$ , Schoof's algorithm produces the following results. First, since  $\lceil 4\sqrt{p} \rceil = 40$ , we need a product of small primes at least this large so the algorithm selects the primes  $\{2, 5, 7\}$ , with  $\prod l_i = 70$ .

Also  $9 < \sqrt{97} < 10$ , so Hasse's theorem gives  $79 \leq \#E(\mathbb{F}_q) \leq 117$ .

The next step is to compute  $t \pmod{2}$ . For this step we find

$$\begin{aligned} x^p \pmod{x^3 + ax + b} &= 47 + 60x + 30x^2 \text{ and} \\ \gcd(x^p - x, x^3 + ax + b) &= 40 + x \end{aligned}$$

Since the gcd is not equal to 1,  $E[2]$  is not empty so  $t \equiv 0 \pmod{2}$ .

Next we test  $\phi_p^2 P = \pm k P$  for  $k \equiv p \pmod{5}$  and we find

$$\begin{aligned} f[5] &= 23 + 67x + 11x^2 + 38x^3 + 77x^4 + 43x^5 \\ &\quad + 93x^6 + 26x^7 + 47x^8 + 87x^9 + 39x^{10} + 5x^{12} \end{aligned}$$

$$\begin{aligned} p_{16} &= 7 + 91x + 40x^2 + 24x^3 + 81x^4 + 69x^5 \\ &\quad + 43x^6 + 45x^7 + 39x^8 + 14x^9 + 30x^{10} + 79x^{11} \end{aligned}$$

$$\gcd(p_{16} f_i) = 1.$$

Hence, there is no point in  $E[5]$  satisfying  $\phi_p^2 P = \pm k P$  so we proceed to case two.

Next we test  $\phi_p^2 + k = \tau \phi_p$  until we find for  $\tau = 2$  that

$$p_{19,x} \equiv 0 \pmod{f_l, p} \text{ and}$$

$$\begin{aligned} \gcd(p_{19,x}, f_l) = & 23 + 67x + 11x^2 + 38x^3 + 77x^4 + 43x^5 \\ & + 93x^6 + 26x^7 + 47x^8 + 87x^9 + 39x^{10} + 5x^{12}. \end{aligned}$$

Since the gcd is not equal to 1, we know that  $\tau = \pm 2$  so we compute

$$\begin{aligned} p_{19,y} = & 39 + 52x + 48x^2 + 33x^3 + 91x^4 + 3x^5 \\ & + 23x^6 + 59x^7 + 16x^8 + 37x^9 + 33x^{10} + 74x^{11} \end{aligned}$$

$$\gcd(p_{19,y}, f_l) = 1.$$

Since this gcd is 1, there is no point in  $E[5]$  satisfying  $\phi_p^2 + k = \tau \phi_p$ , so we must have  $t \equiv -2 \pmod{5} \equiv 3 \pmod{5}$ . Similarly, for  $l = 7$  we find at  $\tau = 3$  that  $\gcd(p_{19,x}, f_7) \neq 1$  and  $\gcd(p_{19,y}, f_7) = 1$  so that  $t \equiv -3 \pmod{7} \equiv 4 \pmod{7}$ . Thus we have the following set of simultaneous congruences.

$$t \equiv 0 \pmod{2}, t \equiv 3 \pmod{5}, t \equiv 4 \pmod{7}.$$

Using the Chinese Remainder theorem we find that the smallest positive integer satisfying this set of congruences is  $t = 18$ . Since  $p + 1 - t = 80$  and 80 is within Hasse's bounds we can conclude  $\#E(\mathbb{F}_p) = 80$ .

## ■ 6.2 Other Experiments

For many other curves our implementation of Schoof's method also produces the correct results, as verified by point counting. For the curve  $E: y^2 = x^3 + 1333x + 1129$  over  $\mathbb{F}_{3571}$ , however, the computed point order is incorrect. The correct order, as determined by direct numerical tests is 3559, giving  $t = 13$ , so that  $t \equiv 1 \pmod{2}$ ,  $t \equiv 2 \pmod{11}$  and  $t \equiv 0 \pmod{13}$ . Our implementation of Schoof's algorithm wrongly determines that  $t \equiv -2 \pmod{11}$ . Analysis of this problem indicates that the most likely source of the error is an incorrect computation for the  $y$ -coordinate part of Schoof equation (19). We are continuing efforts to resolve this issue.

Table 1 summarizes the results of running Schoof's algorithms on several curves. The first column describes the curve  $y^2 = x^3 + Ax + B$  over  $\mathbb{F}_p$  by giving  $A, B, p$ . The second column show the results of computing  $\tau_i \equiv t \pmod{l_i}$  as pairs  $\{\tau_i, l_i\}$ . The third column gives the unique  $t$  within Hasse's bounds that satisfies the



congruences of column two.  $\# E(\mathbb{F}_p)$  is then computed as  $p + 1 - t$ .

**Table 2 - Results from Schoof's Algorithm**

A, B, p	$t \pmod{l_i}$	t	$\# E(\mathbb{F}_p)$	Correct
13, 215, 229	{0, 2}, {0, 5}, {4, 7}	-10	240	yes
106, 166, 197	{0, 2}, {2, 3}, {0, 5}, {1, 11}	-10	208	yes
31, 16, 137	{1, 2}, {0, 3}, {4, 5}, {2, 7}	9	129	yes
503, 367, 523	{1, 2}, {0, 5}, {6, 7}, {7, 11}	-15	539	yes
1333, 1129, 3571	{1, 2}, {9, 11}, {0, 13}	-13	3585	no

### ■ 6.3 Discussion of Results

Schoof's method, as here implemented, has only educational value. This is so for several reasons. First, as noted in references [3,6], the practicality of the algorithm is greatly limited by the quadratic growth of the degree of the division polynomials. For example, for an elliptic curve over  $\mathbb{F}_p$  where  $p$  has 200 digits, we must perform modular polynomial arithmetic using the 55<sup>th</sup> division polynomial, which produces intermediate products of degree greater than  $9 \times 10^6$ . The improvements due to Elkies and Atkin, often called the SEA algorithm, reduce to nearly linear growth the degree of the division polynomials, making the method applicable to elliptic curves of with cryptographic utility. Another limiting factor is the performance of modular polynomial arithmetic in *Mathematica*. Even for the small problems presented in this section, the run-time of our implementation exceeded 10 minutes. Finally, the current implementation is limited by an apparent software bug in the implementation of Schoof equation (19y).

On the other hand, the major advantage of our implementation is as an exploration tool. All of the algorithms we implemented are well-documented, and rely only on low-level functions with *Mathematica*, making the operation of the algorithms transparent and open to experimentation. The author plans to correct the programming deficiencies just cited and to make this suite of elliptic curve functions available as a *Mathematica* package.

## Chapter 7 - Applications

### ■ 7.1 The Elliptic Curve Discrete Log Problem

One way to apply elliptic curve methods to cryptography is via the discrete log problem, which can be described as follows. Suppose  $G$  is a group of finite order and  $g, q$  are elements of  $G$ . Then suppose there exists an integer  $k$  such that  $q = g^k$ . Given the elements  $g, q$  we are asked to find the corresponding exponent  $k$ . Since this looks similar to taking logarithms, we call this the discrete log problem or the DLP. For example, we could choose  $G = \mathbb{F}_q^*$ , the multiplicative group of the finite field  $\mathbb{F}_q$ .

If  $G$  is a finite abelian group and  $g$  is an element of the group, then for any integer  $k$  we define  $k \cdot g = g + g + \dots + g$  (a total of  $k$  times). If  $q = k \cdot g$  and we give someone the two elements  $g, q$  and asks them to find the integer  $k$ , then we have given them the discrete log problem in additive notation. Since the elements of  $E(\mathbb{F}_q)$  form an abelian group we can state the discrete log problem for elliptic curves in the following way. Given two points  $P, Q \in E(\mathbb{F}_q)$  with  $Q = kP$ , find  $k$ . For a given field order  $q$ , this problem is understood to be more difficult than the equivalent problem in  $\mathbb{F}_q^*$ .

### ■ 7.2 Anomalous Curves and the MOV attack

An elliptic curve is called **anomalous** if  $\#E(\mathbb{F}_q) = q$ . When this occurs, the discrete log problem for the group  $E(\mathbb{F}_q)$  can be solved in linear time using an algorithm by Satoh & Araki, outlined in Washington [7] § 5.4.

The MOV attack, first presented by Menezes, Okamoto and Vanstone, uses the Weil pairing to convert the discrete log problem for  $E(\mathbb{F}_q)$  into a discrete log problem for  $\mathbb{F}_{q^s}$ . When  $q$ , the characteristic of the field, divides the trace of the Frobenius  $t$ , then  $s$  will be small. In this case the index calculus can be used to attack the DLP in  $\mathbb{F}_{q^s}$  (Washington [7] §5.3). Hence, when selecting an elliptic curve for cryptographic applications, it is important to determine the order of the resulting group to avoid cases which simplify the discrete log problem for potential eavesdroppers. René Schoof's method, and in particular its extensions, provide a way of efficiently determining  $\#E(\mathbb{F}_q)$  for large integers  $q$ , thus assisting in the creation of secure codes.

## References

- [1] Brown, Ezra - *Square Roots from 1; 24, 51, 10 to Dan Shanks*, College Math Journal (1999).
- [2] Dummit, D., Foote, R. *Abstract Algebra*, John Wiley and Sons (2004)
- [3] Lercier, R. and Morain, F., *Counting the number of points on elliptic curves over finite fields: strategies and performances*, Advances in Cryptology, Proc. Eurocrypt'95, LNCS 921, L.C. Guillou and J.J. Quisquater, Eds., Springer-Verlag, 1995, pp. 79--94.
- [4] Rosen, Kenneth H. - *Elementary Number Theory and its Applications*, Addison and Wesley (2000)
- [5] Schoof, René - *Elliptic curves over finite fields and the computation of square roots mod  $p$* , Mathematics of Computation, Vol. 44, No 170 (1985), 482-494
- [6] Schoof, René - *Counting points on elliptic curves over finite fields*, Journal de Théorie des Nombres de Bordeaux 7 (1995), 219-254
- [7] Washington, Lawrence - *Elliptic Curves - Number Theory and Cryptography*, Chapman & Hall (2003)

## Appendix A - Dictionary of Our *Mathematica* Functions for Elliptic Curves

This section gives a summary of the *Mathematica* functions we developed in order to implement, test and explain Schoof's algorithm. Many of these functions have value independent of Schoof's method and may be useful for other elliptic curve computation projects.

### ■ Number Theoretic Functions

EuclideanAlgorithm[  $a, b$  ]

Compute the greatest common divisor  $d$  of the two integers  $a, b$ .

ExtendedEuclideanAlgorithm[  $a, b$  ]

Compute  $d = \gcd(a, b)$  and return  $\{d, r, s\}$  such that  $d = r a + s b$

MultiplicativeInverse[  $a, p$  ]

Compute  $b$  such that  $a b \equiv 1 \pmod{p}$

ModularExponentiation[  $a, n, p$  ]

Computes  $a^n \pmod{p}$

SquareRootModPShanksTonelli[  $a, p$  ]

Computes  $c$  such that  $c^2 \equiv a \pmod{p}$  if  $a$  is a quadratic residue mod  $p$ .

EcAddMod[  $\{a, b\}, P_1, P_2, p$  ]

Computes  $P_1 + P$  using elliptic curve addition in  $E : y^2 = x^3 + a x + b$  over  $\mathbb{F}_p$ .

EcPowerMod[  $\{a, b\}, Q, k, p$  ]

Computes  $k Q$  in  $E : y^2 = x^3 + a x + b$  over  $\mathbb{F}_p$ .

DetermineChineseRemainder[  $t$  ]

Uses the Chinese Remainder Theorem to solve the set of congruences  $z \equiv r_i \pmod{n_i}$  for  $i = 1, 2, \dots, k$  for the unknown  $z$ , where these congruences are represented by the set of ordered pairs  $t = \{\{r_1, n_1\}, \{r_2, n_2\}, \dots, \{r_k, n_k\}\}$

■ **Elliptic Curve Functions**

`EcAdd[{a, b}, P, Q]`

Add the points  $P, Q$  on the elliptic curve  $E : y^2 = x^3 + ax + b$

`IsEcQ[{a, b}, p]`

Returns true if  $E : y^2 = x^3 + ax + b$  is an elliptic curve, that is, if  $E$  has nonzero discriminant modulo  $p$

`EcPointQ[{a, b}, {x, y}, p]`

Return true if  $(x, y)$  satisfies  $y^2 = x^3 + ax + b$  in  $\mathbb{F}_p$ .

`EcAddMod[{a, b}, P, Q, p]`

Add the points  $P, Q$  on the elliptic curve  $E : y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$

`EcPowerMod[{a, b}, Q, k, p]`

Computes  $kQ$  on the elliptic curve  $E : y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$

`EcPointOrderMod[{a, b}, Q, p, m]`

Determines the value of  $k \leq m$  for which  $kQ = O$ , if any.

`FindEcPointSet[{a, b}, p]`

Determines all points in  $E(\mathbb{F}_p)$  for the curve  $y^2 = x^3 + ax + b$  by exhaustive search.

`ComputeOrderEFq[t, p, n]`

Given that  $\# E(\mathbb{F}_p) = p + 1 - t$ , this method computes  $\# E(\mathbb{F}_{p^n})$  as

$$\# E(\mathbb{F}_{p^n}) = p^n + 1 - (\alpha^n + \beta^n),$$

$$\text{where } X^2 + tX + p = (X - \alpha)(X - \beta)$$

and  $(\alpha^n + \beta^n)$  is computed using a recursion relation

`EcPointOrderBabyGiant[{a, b}, Q, p]`

Computes the order of the point  $Q$  in  $E : y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$  using the Baby Step, Giant Step method.

`GenerateRandomPointEC[{a, b}, p]`

Finds a random point on  $E : y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$

`EcGroupOrder[{a, b}, p]`

Determines  $\# E(\mathbb{F}_p)$  where  $E: y^2 = x^3 + ax + b$ .

It uses `EcPointOrderBabyGiant` on random points in  $E(\mathbb{F}_p)$  until the least common multiple of their orders implies a unique value of the group order within Hasse's bounds.

■ **Functions that Comprise Schoof's Algorithm**

`ComputeTModTwo[ {a, b}, p ]`

Determines  $t(\text{mod } 2)$  where  $\# E(\mathbb{F}_p) = p + 1 - t$ .

`ComputePrimeSet[ p ]`

Determines the set of small primes  $\{l_i\}$  such that  $\sum_{i=1}^L l_i > 4\sqrt{p}$ .

`ComputeDivisionPolynomials[ {a, b}, p, k ]`

Computes the first  $k$  division polynomials for  $E: y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$ .

`ComputeEquation16[ p, l, {a, b} ]`

Computes  $p_{16}(x, y) = (x^{q^2} - x) \psi_k^2 - \psi_{k-1} \psi_{k+1} \pmod{f_l, p}$ .

`ComputeEquation17[ p, l, w, {a, b} ]`

Computes  $p_{17}(x, y) = (x^p - x) \psi_w^2 - \psi_{w-1} \psi_{w+1} \pmod{f_l, p}$ .

`ComputeEquation18[ p, l, w, {a, b} ]`

Computes  $p_{18}(x, y) = 4 \psi_w^3 y^{p+1} - \psi_{w+2} \psi_{w-1}^2 - \psi_{w-2} \psi_{w+1}^2 \pmod{f_l, p}$ .

`ComputeTmodLCaseOne[ p, l, {a, b} ]`

Uses Schoof equations 16, 17, 18 to compute  $t(\text{mod } l)$ .

`ComputeAlpha[ p, l, {a, b} ]`

Given  $\lambda = \alpha / \beta$  for the elliptic curve addition  $\phi_p^2 P + kP$ , compute the polynomial representing  $\alpha$ .

`ComputeBeta[ p, l, {a, b} ]`

Given  $\lambda = \alpha / \beta$  for the elliptic curve addition  $\phi_p^2 P + kP$ , compute the polynomial representing  $\beta$ .

`ComputeEquation19X[ p, l,  $\tau$ , {a, b} ]`

Compute the polynomial, modulo  $f_l$  representing Schoof equation (19x):

$$p_{19_x}(x, y) = \psi_\tau^{2^p} (\beta^2 (\psi_{k-1} \psi_{k+1} - \psi_k^2 (x^{p^2} + x^p + x) + \alpha \psi_k^2)) + \psi_k^2 \beta^2 (\psi_{\tau-1} \psi_{\tau-1})^p$$

ComputeEquation19Y[ p l,  $\tau$ , {a, b} ]

Compute the polynomial, modulo  $f_l$  representing Schoof equation (19y):

$$p_{19_y}(x, y) = 4 f_\tau^{3^p} y^p (((2 x^{p^2} + x) \alpha \beta^2 - \beta^3 y^{p^2} - \alpha^3) f_k^2 - \alpha \beta^2 f_{k-1} f_{k+1}) - \beta^3 f_k^2 (f_{\tau-1}^2 f_{\tau+2} - f_{\tau-2} f_{\tau+1})^p$$

ComputeGroupOrderSchoof[ {a, b}, p ]

Determines  $\# E(\mathbb{F}_p)$  for  $E : y^2 = x^3 + a x + b$  using Schoof's Algorithm

## Appendix B - *Mathematica* Code for Our Elliptic Curve Functions

This appendix provides a listing of *Mathematica* code for each of the functions listed in appendix A.

### ■ Number Theoretic Algorithms

#### ■ EuclideanAlgorithm[ *a*, *b* ]

Compute the greatest common divisor of two integers *a*, *b* using the Euclidean Algorithm.

```
EuclideanAlgorithm[a_, b_] := Module[{x, y, r1, r2, m},
  x = Max[a, b]; y = Min[a, b];
  r1 = y; r2 = Mod[x, y];
  While[r2 > 0,
    m = r2;
    r2 = Mod[r1, r2];
    r1 = m;
  ];
  Return[r1];
];
```

#### ■ ExtendedEuclideanAlgorithm[ *a*, *b* ]

Computes  $d = \gcd(a, b)$  and returns  $\{d, r, s\}$  such that  $d = ra + sb$ . The results are returned as the list  $\{d, r, s\}$ .

```
ExtendedEuclideanAlgorithm[a_Integer, b_Integer] :=
Module[{s1, s2, t1, t2, r1, r2, q1, q2},
  {s1, s2} = {1, 0}; {t1, t2} = {0, 1};
  {r1, r2} = {a, b}; {q1, q2} = {0, 1};
  While[r2 > 0,
    {q1, q2} = {q2, Quotient[r1, r2]};
    {r1, r2} = {r2, Mod[r1, r2]};
    {s1, s2} = {s2, s1 - s2 * q2};
    {t1, t2} = {t2, t1 - t2 * q2};
  ];
  Return[{r1, s1, t1}];
];
```



- **MultiplicativeInverse**[ *a*, *p* ]

The following *Mathematica* function computes the multiplicative inverse of *a* modulo *p*,  $a^{-1} \pmod{p}$ . The function returns  $a^{-1}$  if  $\text{gcd}(a, p) = 1$ , otherwise  $d \mid p$  so that  $d \neq 1$  is a factor of *p* so we throw the message "Factor of *p* found = *d*", which of course means that *p* was not prime.

```
MultiplicativeInverse[a_Integer, p_Integer] := Module[{d, r, s},
  {d, r, s} = ExtendedEuclideanAlgorithm[a, p];
  If[d != 1, Throw[{"Factor of p found = ", d}]];
  r = Mod[r, p];
  Return[r];
];
```

- **ModularExponentiation**[ *a*, *n*, *p* ] returns  $a^n \pmod{p}$

Compute  $a^n \pmod{p}$  - equivalent to the built-in *Mathematica* function `PowerMod`

```
ModularExponentiation[a_Integer, n_Integer, p_Integer] :=
Module[{t, x, b},
  b = n;
  (* set t ≡ a (mod p) in case a < 0 or a > p *)
  t = Mod[a, p];
  x = 1;
  (* for each bit in the binary representation of n *)
  While[b != 0,
    (* if the kth bit is 1 set x ≡ x*ak (mod p) *)
    If[Mod[b, 2] != 0,
      x = Mod[x * t, p];
    ];
    (* a2k ≡ (a2k-1)2 *)
    t = Mod[t * t, p];
    (* bring the next bit
       of n to the least significant position *)
    b = Quotient[b, 2];
  ];
  Return[x];
];
```

- **QuadraticResidueQ**[ *a*, *p* ]

This function uses Euler's criteria, as explained above, to determine if *a* is a quadratic residue mod *p*. It returns true if *a* is a quadratic residue, else it returns false.

```
QuadraticResidueQ[a_, p_] :=
  ModularExponentiation[a, (p - 1) / 2, p] == 1;
```

■ **RandomNonResidue**[  $p$  ]

Choose a random quadratic nonresidue mod  $p$ , that is a number  $1 < x < p$  with  $x^{(p-1)/2} \equiv -1 \pmod{p}$ . Since exactly  $1/2$  of all elements of  $\mathbb{F}_p^\times$  are quadratic non-residues, this will occur within 5 trials with better than 95% probability.

```

RandomNonResidue [p_] := Module[{x},
  x = 3;
  While[QuadraticResidueQ[x, p],
    x = Random[Integer, {2, p - 1}];
  ];
  Return[x];
]

```

■ **SquareRootModPShanksTonelli**[  $a, p$  ]

This function solves the congruence  $x^2 \equiv a \pmod{p}$  for  $x$ , which must be quadratic residue modulo the prime  $p$ . It returns  $x$  if it exists, else it returns  $\{\}$ .

```

SqrtModPShanksTonelli[a_, p_] :=
Module[{s, e, n, q, x, b, g, r, y, m},
  If[! PrimeQ[p],
    Print["ShanksTonelli only works for prime p"];
    Return[{}];
  ];
  If[! QuadraticResidueQ[a, p],
    Print[a, " is not quadratic residue mod ", p];
    Return[{}];
  ];
  (* If p ≡ 3 (mod 4) x ≡ a^{(p+1)/4} (mod p) *)
  If[Mod[p, 4] == 3,
    x = PowerMod[a, (p + 1) / 4, p];
    Return[x];
  ];
  s = p - 1; e = 0;
  (* Compute s, e s p-1 = s*2^e and s is odd *)
  While[EvenQ[s], s = Quotient[s, 2]; ++e;];
  (* Print ["p-1 = ", s, " * 2^", e]; *)
  (* Find n s n^{(p-1)/2} ≡ -1 (mod p) *)
  n = 2;
  q = (p - 1) / 2;
  (* ... this won't take long since
     1/2 elements of Z_p are quadratic nonresidues *)
  While[PowerMod[n, q, p] ≠ p - 1, ++n];
  (* Print ["n = ", n]; *)
  (* Initialize *)
  x = PowerMod[a, (s + 1) / 2, p];
  (* guess at square root x ≡ a^{(s+1)/2} *)
  b = PowerMod[a, s, p]; (* guess at fudge factor b ≡ a^s *)

```

```

g = PowerMod[n, s, p]; (* used to update x, b at each step *)
r = e;
(* exponent, will decrease at each step *)
(* Print["x      b      g      r"]; *)
While[True,
  (* Print[{x,b,g,r}]; *)
  (* Find m < r ∃ b2m ≡ 1 *)
  y = b; m = 0;
  While[y ≠ 1, y = Mod[y*y, p]; ++m];
  If[m == 0, Return[x]];
  (* Compute y = g2r-m-1 *)
  y = g;
  For[i = 1, i ≤ r - m - 1, ++i; y = Mod[y*y, p]];
  (* x = x*g2r-m-1 *)
  x = Mod[x*y, p];
  (* b = b*g2r-m *)
  y = Mod[y*y, p];
  b = Mod[b*y, p];
  (* g = g2r-m, r = m *)
  g = y; r = m;
];
];

```

■ DetermineChineseRemainder[ t ]

Solve the set of integer congruences

$$z \equiv r_i \pmod{n_i} \text{ for } i = 1, 2, \dots, k.$$

for  $z$ , where these congruences are represented by the set of ordered pairs

$$t = \{\{r_1, n_1\}, \{r_2, n_2\}, \dots, \{r_k, n_k\}\}$$

```

DetermineChineseRemainder[t_] := Module[{k, n, r, m, a, e},
  k = Length[t];
  {r, n} = Transpose[t];
  (* Compute N = n1 n2 ... nk *)
  m = Apply[Times, n];
  Print["N = ", m];
  (* Compute ai = N/ni *)
  a = Map[m / # &, n];
  Print["ai = ", a];
  (* Compute bi = ai-1 (mod ni) *)
  b = Inner[PowerMod[#1, -1, #2] &, a, n, List];
  Print["bi = ai-1 (mod ni) = ", b];
  (* Compute ei = ai*bi *)
  e = a * b;
  Print["ei = ai bi = ", e];
  (* Compute z = e.r = e1 r1 + e2 r2 + ... + ek rk *)
  z = e.r;
  Print["z = e.r = ", z];
  Return[Mod[z, m]];
]

```

## ■ Elliptic Curve Arithmetic Algorithms

### ■ EcAdd[ $\{a, b\}, \{x_1, y_1\}, \{x_2, y_2\}$ ]

This *Mathematica* function performs the elliptic curve group addition  $P_1 + P_2$ . The parameters are  $\{A, B\}$  representing  $E: y^2 = x^3 + Ax + B$  and  $P_1 = \{x_1, y_1\}, P_2 = \{x_2, y_2\}$ . Note that the arithmetic in *Mathematica* is exact, so rational input will return a rational sum.

```
EcAdd[ $\{a_, b_\}, \{x1_, y1_\}, \{x2_, y2_\}$ ] := Module[ $\{m, x3, y3\}$ ,
  (* Handle identity cases *)
  If[ $x1 == \infty$ , Return[ $p2$ ]];
  If[ $x2 == \infty$ , Return[ $p1$ ]];
  If[ $x1 == x2 \ \&\& \ y1 == -y2$ , Return[ $\{\infty, \infty\}$ ]];
  (* If we are doubling a point *)
  If[ $x1 == x2 \ \&\& \ y1 == y2$ ,
    (* Compute the slope of the tangent *)
     $m = (3 x1^2 + a) / (2 y1)$ ;
    ,
    (* else compute the slope of the chord *)
     $m = (y2 - y1) / (x2 - x1)$ ;
  ];
   $x3 = m^2 - x1 - x2$ ;
   $y3 = m (x1 - x3) - y1$ ;
  Return[ $\{x3, y3\}$ ];
];
```

### ■ EcDiscriminant[ $\{a, b\}, p$ ]

The discriminant of a polynomial is the product of the squares of the differences between all distinct pairs of roots. For a cubic polynomial this is  $d = (r_1 - r_2)^2 (r_1 - r_3)^2 (r_2 - r_3)^2$ . For the elliptic curve  $y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$  this is given by

$$d \equiv -(4a^3 + 27b^2) \pmod{p}$$

```
EcDiscriminant[ $\{a_, b_\}, p_$ ] :=
  Mod[ $-(4 * \text{ModularExponentiation}[a, 3, p] +$ 
     $27 * \text{ModularExponentiation}[b, 2, p])$ ,  $p$ ];
```

### ■ IsEcQ[ $\{a, b\}, p$ ]

$E: y^2 = x^3 + ax + b$  specifies an elliptic curve  $E(\mathbb{F}_p)$  if and only if the right hand side has distinct roots. This is true if and only if the discriminant is nonzero modulo  $p$ .

```
IsEcQ[ $\{a_, b_\}, p_$ ] := EcDiscriminant[ $\{a, b\}, p] \neq 0$ ;
```

- `EcPointQ[ {a, b}, {x, y}, p ]`

Returns true if  $(x, y) \in E(\mathbb{F}_p)$  if and only if  $y^2 \equiv x^3 + ax + b \pmod{p}$

```
IsEcPointQ[{a_, b_}, {x_, y_}, p_] :=
  Mod[ModularExponentiation[x, 3, p] + a x +
    b - ModularExponentiation[y, 2, p], p] == 0;
```

- `EcAddMod[ {a, b}, P1, P2, p ]`

This function performs the elliptic curve group addition  $P_1 + P_2$  over the finite field  $\mathbb{F}_p$ . The parameters are  $\{a, b\}$ ,  $P_1 = \{x_1, y_1\}$ ,  $P_2 = \{x_2, y_2\}$ , where  $p$  is the characteristic of the field, which must be prime. The function accepts and returns  $\{\infty, \infty\}$  for the group identity  $O$ . It returns  $\{\}$  if either point does not lie on the elliptic curve.

```
EcAddMod[{a_, b_}, p1_, p2_, p_] :=
  Module[{m, x1, y1, x2, y2, x3, y3, w},
    {x1, y1} = p1; {x2, y2} = p2;
    (* Handle identity cases *)
    If[x1 == ∞, Return[p2]];
    If[x2 == ∞, Return[p1]];
    (* P1 + (-P1) = ∞ *)
    If[x1 == x2 && Mod[y1 + y2, p] == 0, Return[{∞, ∞}]];
    (* Verify that the points lie on the curve *)
    If[! IsEcPointQ[{a, b}, {x1, y1}, p], Return[{}]];
    If[! IsEcPointQ[{a, b}, {x2, y2}, p], Return[{}]];
    (* If we are doubling a point *)
    If[p1 == p2,
      (* Check for vertical tangent *)
      If[y1 == 0, Return[{∞, ∞}]];
      (* Compute the slope of the tangent *)
      w = MultiplicativeInverse[2 y1, p];
      m = Mod[(3 x1^2 + a) * w, p];
      ,
      (* else compute the slope of the chord *)
      w = MultiplicativeInverse[x2 - x1, p];
      m = Mod[(y2 - y1) * w, p];
    ];
    x3 = Mod[m^2 - x1 - x2, p];
    y3 = Mod[m (x1 - x3) - y1, p];
    Return[{x3, y3}];
  ];
```

■ **EcPowerMod**[ {a, b}, Q, k, p ]

Compute  $kQ$  in the abelian group of points on the elliptic curve  $E: y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$ . This algorithm is similar to the modular exponentiation method, in that it uses the binary representation of  $k$  to convert the problem into a series of doublings and additions in  $E$ . The algorithm is based on Washington § 2.2 under "Integer Times a Point".

```
EcPowerMod[{a_, b_}, q_, k_, p_] := Module[{i, r, s},
  i = k; r = {∞, ∞}; s = q;
  While[i ≠ 0,
    If[EvenQ[i],
      i = Quotient[i, 2];
      s = EcAddMod[{a, b}, s, s, p];
    ,
      i = i - 1;
      r = EcAddMod[{a, b}, r, s, p];
    ];
  ];
  Return[r];
];
```

■ **EcJinvariant**[ {a, b}, p ]

Computes the  $j$ -invariant of  $E: y^2 = x^3 + ax + b$  as

$$j = 1728 \frac{4a^3}{4a^3 + 27b^3}$$

If two elliptic curves have the same  $j$ -invariant then there exist an isomorphism over the algebraic closure of  $\mathbb{F}_p$  which transforms one into the other.

```
EcJinvariant[{a_, b_}, p_] := Module[{a3, b2, n, d, di, j},
  a3 = PowerMod[a, 3, p];
  b2 = PowerMod[b, 2, p];
  (* denominator is 4 a^3 + 27 b^3 *)
  d = Mod[4 * a3 + 27 * b2, p];
  (* d^-1 (mod p) exists because F_p is a field and d ≠ 0
  for an elliptic curve *)
  di = MultiplicativeInverse[d, p];
  (* j = 1728 * 4 * a^3 / (4 a^3 + 27 b^2) *)
  j = Mod[1728 * 4 * a3 * di, p];
  Return[j];
];
```

■ `EcPointOrderMod[ec, Q, p, m]`

This function computes the order of a point  $Q$  on the elliptic curve  $E(\mathbb{F}_p)$  as the smallest

$$k \in \mathbb{Z}_p \ni kQ = O \text{ on } E(\mathbb{F}_p).$$

It works by direct search so is suitable only if the order is small. It tries orders up to  $m$  and returns zero if  $|Q| > m$

```
EcPointOrderMod[ec_, p1_, p_, m_] := Module[{k, p2, pn3},
  (* Test for special case orders 1,2 *)
  If[p1 == {∞, ∞}, Return[1]];
  p2 = EcAddMod[ec, p1, p1, p];
  If[p2 == {∞, ∞}, Return[2]];
  k = 3;
  (* compute -P (mod p) *)
  pn3 = Mod[p1 * {1, -1}, p];
  (* while (k-1) P ≠ -P *)
  While[p2 ≠ pn3 && k < m,
    (* kP = P + (k-1) P *)
    p2 = EcAddMod[ec, p2, p1, p];
    ++k;
  ];
  (* did not find a solution < m, return 0 *)
  If[k ≥ m, Return[0]];
  (* return the order of P = k *)
  Return[k];
]
```

■ FindEcPointSet[ $\{a, b\}, p$ ]

Find all points in  $E(\mathbb{F}_p)$  by testing every value of  $x \in \mathbb{F}_p$  to determine if there are values of  $y \in \mathbb{F}_p$  satisfying  $E : y^2 = x^3 + Ax + B$ .

```
FindEcPointSet[{a_, b_}, p_] := Module[{s, x, e1, x1, y1, y2},
  e1 = x3 + a x + b;
  s = Reap[
    For[x1 = 0, x1 < p, ++x1,
      y2 = Mod[e1 /. x -> x1, p];
      If[y2 == 0,
        Sow[{x1, 0, 0}];
        ,
        (* If y2 has a square root mod p *)
        If[QuadraticResidueQ[y2, p],
          (* find the smallest y1 s y12 ≡ y2 (mod p) *)
          y1 = SqrtModPShanksTonelli[y2, p];
          Sow[{x1, y1, Mod[-y1, p]}];
        ];
      ];
    ];
  Return[First[Last[s]]];
];
```

■ ProduceEcPointTable[ $\{a, b\}, p$ ]

This function produces a table of all points in  $E(\mathbb{F}_p)$ , along with the order of each point. It returns the order of the group  $\#E(\mathbb{F}_p)$  by counting the points.

```
ProduceEcPointTable[{a_, b_}, p_] := Module[{s, c, t},
  (* Find the set of points on y2 = x3 + a x + b over Fp *)
  s = FindEcPointSet[{a, b}, p];
  (* Compute #E(Fp) by counting the points, including O *)
  c = 2 * Length[s] + 1 - Length[Select[s, Last[#] == 0 &]];
  (* Determine the order of
  each point and put it in the last column *)
  t = Map[Append[#, EcPointOrderMod[{a, b}, Take[#, 2], p, 2 * p]] &,
    s];
  (* Produce a nice table with headings *)
  t2 = Prepend[t, {"x", "\!\(y\ 1\)", "\!\(y\ 2\)", "Ord(P)"}];
  Print[FrameBox[GridBox[t2, RowLines -> True, ColumnLines -> True]] //
    DisplayForm];
  Return[c];
];
```



## ■ Methods to Determine the Elliptic Curve Group Order

### ■ ComputeOrderEFq[ $t, p, n$ ]

Given that  $\#E[\mathbb{F}_p] = p + 1 - t$ , this method computes and returns  $\#E[\mathbb{F}_{p^n}]$ .

```

ComputeOrderEFq[t_, p_, n_] := Module[{sn0, sn1, sn2, k},
  sn0 = 2;
  sn1 = t;
  For[k = 2, k ≤ n, ++k,
    sn2 = t * sn1 - p * sn0;
    sn0 = sn1;
    sn1 = sn2;
  ];
  Return[p^n + 1 - sn2];
];

```

### ■ EcPointOrderBabyGiant[ $\{a, b\}, Q, p$ ]

Use the BabyStep-GiantStep method to find order of the point  $Q$ , that is find  $m \in \mathbb{Z}_q \ni mQ = \infty$  on the elliptic curve  $E: y^2 = x^3 + ax + b$ . This method is described in Washington[7] § 4.3.

```

EcPointOrderBabyGiant[ec_, P1_, p_] :=
Module[{Q1, i, j, k, m, b, t1, t2, p2,
  pm3, pm4, m1, f1, x1, q2, q3, pn5, pn6, pn7},
  (* Check for small order directly *)
  k = EcPointOrderMod[ec, P1, p, Min[100, p]];
  If[k > 0, Return[k]];
  (* Compute Q = (p+1)P *)
  Q1 = EcPowerMod[ec, P1, p + 1, p];
  (* Table size m = ⌈√[4]{p}⌉ *)
  m = Ceiling[Power[p, 1/4]];
  (* Create table of iP for i=1,m *)
  t1 = {P1};
  p2 = P1;
  For[i = 2, i ≤ m, ++i,
    p2 = EcAddMod[ec, p2, P1, p];
    AppendTo[t1, p2];
  ];
  t2 = First[Transpose[t1]];
  q2 = Q1;
  q3 = Q1;
  (* Create table of Q + 2kmP for k=-m,...,m*)
  pm3 = EcPowerMod[ec, P1, 2 * m, p];
  pm4 = pm3;
  (* Compute -P (mod p) *)
  pn5 = Mod[P1 * {1, -1}, p];

```

```

pn6 = EcPowerMod[ec, pn5, 2 * m, p];
pn7 = pn6;
k = 0;
(* Compute for k=±1,±2,... *)
While[True,
  (* Until Q + 2 kmP = ±jP or ... *)
  If[MemberQ[t2, First[q2]],
    Break[];
  ];
  (* Until Q - 2 kmP = ±jP or ... *)
  If[MemberQ[t2, First[q3]],
    k = -k;
    q2 = q3;
    Break[];
  ];
  (* Compute next Q ± 2 kmP for next k *)
  q2 = EcAddMod[ec, Q1, pm4, p];
  q3 = EcAddMod[ec, Q1, pn7, p];
  pm4 = EcAddMod[ec, pm4, pm3, p];
  pn7 = EcAddMod[ec, pn7, pn6, p];
  ++k;
];
(* Here we have Q ± 2 kmP = ±jP, figure out j *)
j = First[Flatten[Position[t2, First[q2]]]];
(* Determine which ±j *)
p2 = t1[[j]];
If[Last[q2] ≠ Last[p2],
  If[Last[q2] == Mod[-Last[p2], p],
    j = -j;
  ],
  Print["ERROR: Q + kmP ≠ ±jP"];
  Return[0];
];
];
(* Compute M ≅ Order(P) | M *)
m1 = p + 1 + 2 * m * k - j;
(* TBD - This fails if m1 = 0 => j = p + 1 + 2*m*k *)
If[m1 == 0, Print["ERROR - M == 0"]; Return[0]];
(* If M is prime, it must be the order of P *)
If[PrimeQ[m1], Return[m1]];
(* Factor can be VERY hard if no small factors *)
f1 = FactorInteger[m1];
b = True;
While[b,
  b = False;
  For[i = 1, i ≤ Length[f1], ++i,
    (* Divide out a prime factor *)
    x1 = m1 / First[f1[[i]]];
    (* If x P (mod p) = P, M = x *)
    pm3 = EcPowerMod[ec, P1, x1 - 1, p];
    If[pm3 == pn5,
      m1 = x1;

```

```

    b = True;
    If[--f1[[i, 2]] ≤ 0, f1 = Delete[f1, i]];
    Break[];
  ];
];
];
Return[m1];
];

```

■ **GenerateRandomPointEC**[{a, b}, p]

Generate a random point  $\{x, y\}$  on the elliptic curve given by  $ec = \{A, B\}$  such that

$$y^2 = x^3 + Ax + B \text{ with all arithmetic modulo } p.$$

```

GenerateRandomPointEC[{a_, b_}, p_] := Module[{x, e1, x1, y1, y2},
  e1 = x3 + a x + b;
  y2 = 1;
  While[True,
    x1 = Random[Integer, {1, p - 1}];
    y2 = Mod[e1 /. x → x1, p];
    (* If y2 has a square root mod p *)
    If[QuadraticResidueQ[y2, p],
      (* find the smallest y1 ≡ y12 ≡ y2 (mod p) *)
      y1 = SqrtModPShanksTonelli[y2, p];
      Break[];
    ];
  ];
  Return[{x1, y1}];
];

```

■ **GenerateRandomEC**[p]

Generate a random elliptic curve over the field  $\mathbb{F}_p$  where  $p$  is a prime greater than 3.

The curve is of the form  $y^2 = x^3 + Ax + B$  with  $4A^2 + 27B^3 \neq 0$

Returns {A,B}

```

GenerateRandomEC[p_] := Module[{a, b, e},
  a = 0; b = 0; e = False;
  While[! e,
    a = Random[Integer, {1, p - 1}];
    b = Random[Integer, {0, p - 1}];
    e = IsEcQ[{a, b}, p];
  ];
  Return[{a, b}];
];

```

■ `EcGroupOrder[ {a, b}, p ]`

This function determines  $\#E(\mathbb{F}_p)$  by determining the order of random points on the curve until there is only one multiple of the least common multiple of these orders within Hasse's bounds.

```

EcGroupOrder[ec_, p_] :=
Module[{pr2, h0, h1, p1, p2, k1, k2, or1, or3, t, k},
  pr2 = Floor[2 * Sqrt[p]];
  h0 = p + 1 - pr2; h1 = p + 1 + pr2;
  Print["Hasse's bounds ", h0, " ≤ #E(Fp) ≤ ", h1];
  or1 = 0;
  t = {};
  While[True,
    (* Generate a random point on E(Fp) *)
    p1 = GenerateRandomPointEC[ec, p];
    Print["P = ", p1];
    (* Determine |p1| *)
    k1 = EcPointOrderBabyGiant[ec, p1, p];
    Print["P has order = ", k1];
    If[k1 ≠ 0,
      t = Union[t, {k1}];
    ];
    (* Compute the LCM of all orders found *)
    or1 = Apply[LCM, t];
    Print["LCM = ", or1];
    (* There is a solution if  $p + 1 - 2\sqrt{p} \leq$ 
      LCM  $\leq p + 1 + 2\sqrt{p}$  *)
    If[h0 ≤ or1 && or1 ≤ h1, Return[or1]];
    (* Determine # of
      multiples of the LCM within Hasse's bounds *)
    k2 = Quotient[h1, or1] - Quotient[h0, or1];
    (* If there is only one multiple of the LCM within
      Hasse's bounds, it must be the order of the group *)
    If[k2 == 1,
      or3 = Quotient[h0, or1] * or1;
      If[or3 < h0, or3 += or1];
      Return[or3];
    ],
    Print["There are ", k2,
      " multiples of the LCM within Hasses's bounds"];
  ];
];
Return[{}];
];

```

## ■ The Functions that Comprise Schoof's Algorithm

### ■ ComputeTModTwo[ {a, b}, p ]

Compute  $t \pmod{2}$  by determining if  $E: y^2 = x^3 + ax + b$  over  $\mathbb{F}_p$  has any points of order 2.

```

ComputeTModTwo[{a_, b_}, p_] := Module[{y2, xp, g},
  y2 = x3 + a x + b;
  (* Compute xp (mod x3 + a x + b, p) *)
  xp = PolynomialPowerMod[x, p, {y2, p}];
  Print["xp (mod x3 + a x + b) = ", xp];
  (* If GCD(xp - x, x3 + a x + b) = 1 then t ≡ 1 (mod 2),
     else t ≡ 0 (mod 2) *)
  g = PolynomialGCD[xp - x, y2, Modulus → p];
  Print["gcd(xp - x, x3 + a x + b) = ", g];
  If[SameQ[g, 1], Return[1], Return[0]];
]

```

### ■ ComputePrimeSet[ p ]

Creates a set  $S$  of the first  $k$  prime numbers  $l_i$  with the smallest  $k$  such that

$$M = \prod_{i=1}^k l_i > 4\sqrt{p} \quad \text{with } p \pmod{l_i} > 1$$

and returns  $\{M, \{l_i\}\}$ .

```

ComputePrimeSet[p_] := Module[{i, x, m, r, s},
  x = 2; (* l1 *)
  m = x; (* M = ∏ li *)
  r = 4 * Ceiling[Sqrt[p]];
  Print["⌈4√p⌉ = ", r];
  s = Reap[
    Sow[x];
    While[m ≤ r,
      x = NextPrime[x];
      (* Exclude li if p ≡ 1 mod li *)
      If[Mod[p, x] > 1,
        m *= x;
        Sow[x];
      ];
    ];
  ];
  Print["∏ li = ", m];
  Return[{m, Flatten[Last[s]]}];
]

```

### ■ Mathematica Code to generate the plot

■ `ComputeDivisionPolynomials[ {a, b}, p, k ]`

Compute the first  $k$  division polynomials for the elliptic curve  $E : y^2 = x^3 + Ax + B$  over  $\mathbb{F}_p$ .

```

ComputeDivisionPolynomials[{a_, b_}, p_, k_] := Module[{t, i, d, m},
  t = {-1, 0, 1, 2 y};
  d = 3 x^4 + 6 a x^2 + 12 b x - a^2;
  d = PolynomialMod[d, p];
  d = Collect[d, {x}];
  AppendTo[t, d];
  d = 4 y (x^6 + 5 a x^4 + 20 b x^3 - 5 a^2 x^2 - 4 a b x - 8 b^2 - a^3);
  d = PolynomialMod[d, p];
  d = Collect[d, {y, x}];
  AppendTo[t, d];
  For[i = 5, i ≤ k + 4, ++i,
    If[EvenQ[i],
      (* Even case *)
      m = 2 + i / 2;
      d = t[[m]] *
        (t[[m + 2]] * (t[[m - 1]])^2 - t[[m - 2]] * (t[[m + 1]])^2) / (2 y);
    ,
      (* Odd case *)
      m = 2 + (i - 1) / 2;
      d = t[[m + 2]] * (t[[m]])^3 - t[[m - 1]] * (t[[m + 1]])^3;
    ];
    (* Replace all y^2 with x^3 + a x + b *)
    d = d /. y^w_ -> (x^3 + a x + b)^(w/2) /; EvenQ[w];
    d = d /. y^w_ -> y (x^3 + a x + b)^(w-1)/2 /; OddQ[w];
    d = PolynomialMod[d, p];
    d = Collect[d, {y, x}];
    AppendTo[t, d];
  ];
  Return[t];
];

```

- $f[n]$  - Compute the  $n$ 'th division polynomial in  $f_n(x)$

We have the following relation

$$f_n(x) = \begin{cases} \psi_n(x, y) & \text{if } n \text{ is odd} \\ \psi_n(x, y)/y & \text{if } n \text{ is even} \end{cases}$$

NOTE:  $f[n]$ ,  $\psi[n]$  rely on the global variable  $psi$  containing the raw information for the division polynomials. This can be accomplished by executing the following statement before the first call to  $f[n]$  or  $\psi[n]$ .

```
psi = ComputeDivisionPolynomials[{a, b}, p, k]
```

```
f[n_] := Module[{},
  If[n < -1 || n > (Length[psi] - 2),
    Throw["f[n] - Invalid index"]];
  Return[psi[[n + 2]] /. y -> 1];
];

psi[n_] := psi[[n + 2]];
```

■ `ComputeEquation16[p, l, {a, b}]`

Computes  $p_{16}(x, y) = (x^{q^2} - x) \psi_k^2 - \psi_{k-1} \psi_{k+1} \pmod{f_l, p}$  in order to test if  $\phi_l^2 P = \pm q P$  for some  $P \in E[l]$ .

```

ComputeEquation16[p_, l_, {a_, b_}] :=
Module[{k, f1, y2, p1, p2, p3, p16x},
  f1 = f[l];
  k = Mod[p, l];
  y2 = x^3 + a x + b;
  (* p1 = x^{q^2} (mod f1) *)
  p1 = PolynomialPowerMod[x, p, {f1, p}];
  p1 = PolynomialPowerMod[p1, p, {f1, p}];
  (* p2 = (x^{q^2} - x) f_k^2(x) (mod f1) *)
  p2 = PolynomialPowerMod[f[k], 2, {f1, p}];
  p2 = PolynomialPowerMod[(p1 - x) * p2, 1, {f1, p}];
  (* p3 = f_{k-1}(x) f_{k+1}(x) *)
  p3 = f[k - 1] * f[k + 1];
  If[EvenQ[k],
    (* p16 = (x^{q^2} - x) f_k^2(x) y^2 + f_{k-1}(x) f_{k+1}(x) k even *)
    p16x = p2 * y2 + p3;
  ,
    (* p16 = (x^{q^2} - x) f_k^2(x) + f_{k-1}(x) f_{k+1}(x) y^2 k odd *)
    p16x = p2 + p3 * y2;
  ];
  p16x = PolynomialPowerMod[p16x, 1, {f1, p}];
  Return[p16x];
];

```



■ **ComputeEquation17**[ p, l, w, {a, b} ]

Given that  $\phi_l^2 P = \pm q P$  for some  $P \in E[l]$ , then  $t \in \{0, -2w, 2w\}$  where  $w^2 \equiv q \pmod{l}$ . If  $q$  has no square root mod  $l$  then we must have  $t = 0$ .

Otherwise we find  $w \ni w^2 \equiv q \pmod{l}$  and use **Schoof (17)** to test if  $\phi_l P = \pm w P$

$$p_{17}(x) = (x^q - x) f_w^2(x) (x^3 + ax + b) + f_{w-1}(x) f_{w+1}(x) \quad w \text{ even}$$

$$p_{17}(x) = (x^q - x) f_w^2(x) + f_{w-1}(x) f_{w+1}(x) (x^3 + ax + b) \quad w \text{ odd}$$

If  $\gcd(p_{17}(x), f_l(x)) = 1$  then neither  $w$  nor  $-w$  is an eigenvalue of  $\phi_l$  so  $t \equiv 0 \pmod{l}$

```

ComputeEquation17[p_, l_, w_, {a_, b_}] :=
Module[{y2, f1, p1, p2, p3, p17},
  y2 = x3 + a x + b;
  f1 = f[l];
  (* p2 = (xq-x) f_w^2(x) *)
  p1 = PolynomialPowerMod[x, p, {f1, p}];
  p2 = (p1 - x) * PolynomialPowerMod[f[w], 2, {f1, p}];
  (* p3 = f_{w-1}(x) f_{w+1}(x) *)
  p3 = f[w - 1] * f[w + 1];
  If[EvenQ[w],
    (* p(x) = (xq-x) f_w^2(x) y^2 + f_{w-1}(x) f_{w+1}(x) *)
    p17 = p2 * y2 + p3;
  ,
    (* p(x) = (xq-x) f_w^2(x) + f_{w-1}(x) f_{w+1}(x) y^2 *)
    p17 = p2 + p3 * y2;
  ];
  p17 = PolynomialPowerMod[p17, 1, {f1, p}];
  Return[p17];
];

```

■ **ComputeEquation18**[ p, l, w, {a, b} ]

After we know from using **Schoof(17)** that  $\phi_l P = \pm w P$  we use **Schoof (18)** to test the y coordinate of  $\phi_l P = w P$ .

For w even:

$$p_{18}(x) = 4 (y^2)^{(q+3)/2} f_w^3(x) - f_{w+2}(x) f_{w-1}^2(x) + f_{w-2}(x) f_{w+1}^2(x)$$

For w odd:

$$p_{18}(x) = 4 (y^2)^{(q-1)/2} f_w^3(x) - f_{w+2}(x) f_{w-1}^2(x) + f_{w-2}(x) f_{w+1}^2(x)$$

```

ComputeEquation18[p_, l_, w_, {a_, b_}] :=
Module[{y2, f1, k, p1, p2, p3, p4, p18y},
  y2 = x3 + a x + b;
  f1 = f[l];
  (* k = (p+3)/2 (w even), k = (p-1)/2 (w odd) *)
  If[EvenQ[w], k = (p + 3) / 2;, k = (p - 1) / 2];
  (* p1 = (x3 + a x + b)k = (y2)k *)
  p1 = PolynomialPowerMod[y2, k, {f1, p}];
  (* p2 = f_w3(x) *)
  p2 = PolynomialPowerMod[f[w], 3, {f1, p}];
  (* p3 = f_{w+2}(x) f_{w-1}^2(x) *)
  p3 = PolynomialPowerMod[f[w - 1], 2, {f1, p}];
  p3 = PolynomialPowerMod[f[w + 2] * p3, 1, {f1, p}];
  (* p4 = f_{w-2}(x) f_{w+1}^2(x) *)
  p4 = PolynomialPowerMod[f[w + 1], 2, {f1, p}];
  p4 = PolynomialPowerMod[f[w - 2] * p4, 1, {f1, p}];
  (* p18y = 4 y2k f_w3(x) - f_{w+2}(x) f_{w-1}^2(x) + f_{w-2}(x) f_{w+1}^2(x) *)
  p18y = 4 * p1 * p2 - p3 + p4;
  p18y = PolynomialPowerMod[p18, 1, {f1, p}];
  Return[p18y];
];

```

■ `ComputeTmodLCaseOne[ p, l, {a, b} ]`

Compute  $t \pmod{l}$  for the case when there exists  $P \in E[l]$  such that  $\phi_l^2 P = \pm k P$ . This method uses Schoof equations (17) and (18) to perform these tests.

```

ComputeTmodLCaseOne [p_, l_, {a_, b_}] :=
Module[{j, t1, w, f1, p17, p18},
(* If p is a quadratic nonresidue of l
then t ≡ 0 (mod l) *)
If[! QuadraticResidueQ[p, l], Return[0]];
(* Else find w such that w^2 ≡ p (mod l) *)
w = SqrtModPShanksTonelli[p, l];
Print["Sqrt of q mod l = ", w];
f1 = f[l];
Print["f[" , l, "] = ", f1];
(* Use Schoof 17 to test ϕ_l P = ±w*P (x coord) *)
p17 = ComputeEquation17[p, l, w, {a, b}];
Print["p17 = ", p17];
g = PolynomialGCD[p17, f1, Modulus → p];
Print["gcd(p17, f1) = ", g];
(* g = 1 means no such P, so t ≡ 0 (mod l) *)
If[SameQ[g, 1], Return[0]];
(* Use Schoof 18 to test ϕ_l P = w*P (y coord) *)
p18 = ComputeEquation18[p, l, w, {a, b}];
Print["p18 = ", p18];
g = PolynomialGCD[p18, f1, Modulus → p];
Print["gcd(p18, f1) = ", g];
(* g = 1 means no such P, so t = -2w, else t = 2w *)
If[SameQ[g, 1], t1 = Mod[-2 w, l], t1 = Mod[2 w, l]];
Return[t1];
];

```

■ **ComputeAlpha**[ p, l, {a, b} ]

**ComputeAlpha**[p, l, {a, b}]

Compute  $\alpha$ , where  $\lambda = \frac{\alpha}{\beta}$  is the slope of the line between  $\phi_l^2 P$  and  $k P$

$$\alpha = \psi_{k+2} \psi_{k-1}^2 - \psi_{k-2} \psi_{k+1}^2 - 4 y^{p^2+1} \psi_k^3$$

for  $k$  even

$$\alpha = y (f_{k+2} f_{k-1}^2 - f_{k-2} f_{k+1}^2 - 4 (y^2)^{(p^2+3)/2} f_k^3)$$

however we return  $\alpha/y$  in this case and compensate at a higher level

for  $k$  odd

$$\alpha = y^2 (f_{k+2} f_{k-1}^2 - f_{k-2} f_{k+1}^2) - 4 (y^2)^{(p^2+1)/2} f_k^3$$

```

ComputeAlpha[p_, l_, {a_, b_}] :=
Module[{k, f1, y2, y4, p1, p2, p3, p4, p5, alpha},
  k = Mod[p, l];
  f1 = f[l];
  (* y^2, y^4 are polynomials in x *)
  y2 = x^3 + a x + b;
  y4 = PolynomialPowerMod[y2, 2, {f1, p}];
  (* p1 = f_{k+2} f_{k-1}^2 *)
  p1 = PolynomialPowerMod[f[k - 1], 2, {f1, p}];
  p1 = PolynomialPowerMod[f[k + 2] * p1, 1, {f1, p}];
  (* p2 = f_{k-2} f_{k+1}^2 *)
  p2 = PolynomialPowerMod[f[k + 1], 2, {f1, p}];
  p2 = PolynomialPowerMod[f[k - 2] * p2, 1, {f1, p}];
  (* p3 = (y^2)^{(p^2-1)/2} = (y^2)^{(p-1)(p+1)/2} *)
  p3 = PolynomialPowerMod[y2, (p - 1), {f1, p}];
  p3 = PolynomialPowerMod[p3, (p + 1) / 2, {f1, p}];
  (* p4 = f_k^3 *)
  p4 = PolynomialPowerMod[f[k], 3, {f1, p}];
  If[EvenQ[k],
    (* alpha = y (f_{k+2} f_{k-1}^2 - f_{k-2} f_{k+1}^2 - 4 (y^2)^{(p^2+3)/2} f_k^3) / y *)
    p5 = PolynomialPowerMod[4 * y4 * p3 * p4, 1, {f1, p}];
    alpha = PolynomialPowerMod[p1 - p2 - p5, 1, {f1, p}];
  ,
    (* alpha = y^2 (f_{k+2} f_{k-1}^2 - f_{k-2} f_{k+1}^2) - 4 (y^2)^{(p^2+1)/2} f_k^3 *)
    p5 = PolynomialPowerMod[4 * y2 * p3 * p4, 1, {f1, p}];
    alpha = PolynomialPowerMod[y2 * (p1 - p2) - p5, 1, {f1, p}];
  ];
Return[alpha];
]

```

■ **ComputeBeta**[ p, l, {a, b} ]

**ComputeBeta**[p, l, {a, b}]

Compute  $\beta$ , where  $\lambda = \frac{\alpha}{\beta}$  is the slope of the line between  $\phi_l^2 P$  and  $k P$

$$\beta = 4 y \psi_k ((x - x^{p^2}) \psi_k^2 - \psi_{k-1} \psi_{k+1})$$

for  $k$  even

$$\beta = 4 y^2 f_k ((x - x^{p^2}) y^2 f_k^2 - f_{k-1} f_{k+1})$$

for  $k$  odd

$$\beta = 4 y f_k ((x - x^{p^2}) f_k^2 - y^2 f_{k-1} f_{k+1})$$

however we return  $\beta/y$  in this case and compensate at a higher level

```

ComputeBeta[p_, l_, {a_, b_}] :=
Module[{k, f1, y2, p1, p2, p3, p4, beta},
  k = Mod[p, l];
  f1 = f[l];
  (* y^2 is a polynomial in x *)
  y2 = x^3 + a x + b;
  (* p1 = x - x^{p^2} *)
  p1 = PolynomialPowerMod[x, p, {f1, p}];
  p1 = PolynomialPowerMod[p1, p, {f1, p}];
  p1 = x - p1;
  (* p2 = f_{k-1} f_{k+1} *)
  p2 = PolynomialPowerMod[f[k - 1] * f[k + 1], 1, {f1, p}];
  (* p3 = f_k^2 *)
  p3 = PolynomialPowerMod[f[k], 2, {f1, p}];
  If[EvenQ[k],
    (* beta = 4 y^2 f_k ((x - x^{p^2}) y^2 f_k^2 - f_{k-1} f_{k+1}) *)
    p4 = PolynomialPowerMod[p1 * y2 * p3 - p2, 1, {f1, p}];
    beta = PolynomialPowerMod[4 * y2 * f[k] * p4, 1, {f1, p}];
  ,
    (* beta = 4 y f_k ((x - x^{p^2}) f_k^2 - y^2 f_{k-1} f_{k+1}) / y *)
    p4 = PolynomialPowerMod[p1 * p3 - y2 * p2, 1, {f1, p}];
    beta = PolynomialPowerMod[4 * f[k] * p4, 1, {f1, p}];
  ];
  Return[beta];
]

```

■ ComputeEquation19X[ p, l,  $\tau$ , {a, b} ]

Compute the polynomial, modulo  $f_l$  representing Schoof equation (19x) as

$$p_{19_x}(x, y) = \psi_\tau^{2^p} (\beta^2 (\psi_{k-1} \psi_{k+1} - \psi_k^2 (x^{p^2} + x^p + x) + \alpha \psi_k^2)) + \psi_k^2 \beta^2 (\psi_{\tau-1} \psi_{\tau-1})^p$$

```

ComputeEquation19X[p_, l_,  $\tau$ _, {a_, b_}] := Module[
  {y2, f1, k,  $\alpha$ ,  $\beta$ ,  $\alpha^2$ ,  $\beta^2$ , p1, p2, p3, p4, p5, p6, p7, p8, p9, p19},
  k = Mod[p, l];
  f1 = f[l];
  (*  $y^2$  is a polynomial in x *)
  y2 = x3 + a x + b;
  (*  $\lambda = \alpha/\beta$ ,
  note these are actually  $\alpha/y$  for k even,  $\beta/y$  for k odd *)
  (* Note  $\alpha, \beta$  are globals so we can share
  them with Equation19Y *)
   $\alpha$  = ComputeAlpha[p, l, {a, b}];
   $\beta$  = ComputeBeta[p, l, {a, b}];
  (* p1 =  $f_{k-1} f_{k+1}$  *)
  p1 = f[k - 1] * f[k + 1];
  (* p2 =  $x^p$  *)
  p2 = PolynomialPowerMod[x, p, {f1, p}];
  (* p3 =  $x^{p^2}$  *)
  p3 = PolynomialPowerMod[p2, p, {f1, p}];
  (* p4 =  $(x^{p^2} + x^p + x)$  *)
  p4 = p3 + p2 + x;
  (* p5 =  $f_k^2$  *)
  p5 = PolynomialPowerMod[f[k], 2, {f1, p}];
  (* p6 =  $(f_{\tau-1} f_{\tau+1})^p$  *)
  p6 = PolynomialPowerMod[f[ $\tau$  - 1] * f[ $\tau$  + 1], p, {f1, p}];
  (* Compute  $\alpha^2, \beta^2$  compensating for single y's *)
  If[EvenQ[k],
    (* for k even  $\alpha^2 = y^2 (\alpha/y)^2$  *)
     $\alpha^2$  = y2 * PolynomialPowerMod[ $\alpha$ , 2, {f1, p}];
     $\beta^2$  = PolynomialPowerMod[ $\beta$ , 2, {f1, p}];
    (* p7 =  $(f_{k-1} f_{k+1} - y^2 f_k^2 (x^{p^2} + x^p + x)) \beta^2 + y^2 f_k^2 \alpha^2$  *)
    p7 = PolynomialPowerMod[p1 - y2 * p5 * p4, 1, {f1, p}];
    p7 = PolynomialPowerMod[p7 *  $\beta^2$  + y2 * p5 *  $\alpha^2$ , 1, {f1, p}];
  ,
    (* for k odd  $\beta^2 = y^2 (\beta/y)^2$  *)
     $\alpha^2$  = PolynomialPowerMod[ $\alpha$ , 2, {f1, p}];
     $\beta^2$  = y2 * PolynomialPowerMod[ $\beta$ , 2, {f1, p}];
    (* p7 =  $(y^2 f_{k-1} f_{k+1} - f_k^2 (x^{p^2} + x^p + x)) \beta^2 + f_k^2 \alpha^2$  *)
    p7 = PolynomialPowerMod[y2 * p1 - p5 * p4, 1, {f1, p}];
    p7 = PolynomialPowerMod[p7 *  $\beta^2$  + p5 *  $\alpha^2$ , 1, {f1, p}];
  ];
  If[EvenQ[k] && EvenQ[ $\tau$ ],
    (*  $((f_{k-1} f_{k+1} - y^2 f_k^2 (x^{p^2} + x^p + x)) \beta^2 + y^2 f_k^2 \alpha^2) y^{2p} f_\tau^{2p} +$ 
     $f_{\tau-1}^p f_{\tau+1}^p \beta^2 y^2 f_k^2$  *)

```

```

p8 = PolynomialPowerMod[f[τ], 2 * p, {f1, p}];
p9 = PolynomialPowerMod[y2, p, {f1, p}];
p19 = PolynomialPowerMod[p7 * p9 * p8 + p6 * β2 * y2 * p5, 1, {f1, p}];
];
If[EvenQ[k] && OddQ[τ],
(* ((f_{k-1} f_{k+1} - y^2 f_k^2 (x^{p^2} + x^p + x)) β^2 + y^2 f_k^2 α^2) f_τ^{2p} +
y^{2p+2} f_{τ-1}^p f_{τ+1}^p β^2 f_k^2 *)
p8 = PolynomialPowerMod[f[τ], 2 * p, {f1, p}];
p9 = PolynomialPowerMod[y2, p + 1, {f1, p}];
p19 = PolynomialPowerMod[p7 * p8 + p9 * p6 * β2 * p5, 1, {f1, p}];
];
If[OddQ[k] && EvenQ[τ],
(* ((y^2 f_{k-1} f_{k+1} - f_k^2 (x^{p^2} + x^p + x)) β^2 + f_k^2 α^2) y^{2p} f_τ^{2p} +
f_{τ-1}^p f_{τ+1}^p β^2 f_k^2 *)
p8 = PolynomialPowerMod[f[τ], 2 * p, {f1, p}];
p9 = PolynomialPowerMod[y2, p, {f1, p}];
p19 = PolynomialPowerMod[p7 * p9 * p8 + p6 * β2 * p5, 1, {f1, p}];
];
If[OddQ[k] && OddQ[τ],
(* ((y^2 f_{k-1} f_{k+1} - f_k^2 (x^{p^2} + x^p + x)) β^2 + f_k^2 α^2) f_τ^{2p} +
y^{2p} f_{τ-1}^p f_{τ+1}^p β^2 f_k^2 *)
p8 = PolynomialPowerMod[f[τ], 2 * p, {f1, p}];
p9 = PolynomialPowerMod[y2, p, {f1, p}];
p19 = PolynomialPowerMod[p7 * p8 + p9 * p6 * β2 * p5, 1, {f1, p}];
];
Return[p19];
];

```

■ `ComputeEquation19Y[ p, l,  $\tau$ , {a, b} ]`

Compute the polynomial, modulo  $f_l$  representing Schoof equation (19y) as

$$p_{19y}(x, y) = 4 f_{\tau}^3 y^p (((2 x^{p^2} + x) \alpha \beta^2 - \beta^3 y^{p^2} - \alpha^3) f_k^2 - \alpha \beta^2 f_{k-1} f_{k+1}) - \beta^3 f_k^2 (f_{\tau-1}^2 f_{\tau+2} - f_{\tau-2}^2 f_{\tau+1}^2)^p$$

```

ComputeEquation19Y[q_, l_,  $\tau$ _, {a_, b_}] :=
Module[{y2, f1, k,  $\alpha$ ,  $\beta$ , a2, a3, b2,
  b3, y4, yq, p1, p2, p3, p4, p5, p6, p19},
  f1 = f[l];
  k = Mod[q, l];
  (* Make sure we have no junk floating around *)
  Print[{q, l,  $\tau$ , x}];
  (*  $y^2$  is a polynomial in x only *)
  y2 = x^3 + a x + b;
  (*  $\lambda = \alpha/\beta$ ,
  note these are actually  $\alpha/y$  for k even,  $\beta/y$  for k odd *)
  (* Use the  $\alpha, \beta$  computed for Equation 19 X *)
   $\alpha$  = ComputeAlpha[q, l, {a, b}];
   $\beta$  = ComputeBeta[q, l, {a, b}];
  (* Compute  $\alpha^3, \beta^2, \beta^3$  *)
  a3 = PolynomialPowerMod[ $\alpha$ , 3, {f1, q}];
  b2 = PolynomialPowerMod[ $\beta$ , 2, {f1, q}];
  b3 = PolynomialPowerMod[ $\beta$ , 3, {f1, q}];
  (*  $p_1 = \alpha \beta^2 (f_{k-1} f_{k+1})$  *)
  p1 =  $\alpha$  * b2 * f[k - 1] * f[k + 1];
  p1 = PolynomialPowerMod[p1, 1, {f1, q}];
  (*  $p_2 = f_k^2$  *)
  p2 = PolynomialPowerMod[f[k], 2, {f1, q}];
  (*  $p_3 = 2 x^{q^2} + x$  *)
  p3 = PolynomialPowerMod[x, q, {f1, q}];
  p3 = PolynomialPowerMod[p3, q, {f1, q}];
  p3 = PolynomialPowerMod[2 * p3 + x, 1, {f1, q}];
  (*  $p_4 = 4 f_{\tau}^{3q} = (4 f_{\tau}^3)^q$  *)
  p4 = 4 * PolynomialPowerMod[f[ $\tau$ ], 3 q, {f1, q}];
  (*  $p_5 = (f_{\tau-1}^2 f_{\tau+2} - f_{\tau-2}^2 f_{\tau+1}^2)^q$  *)
  p5 = f[ $\tau$  + 2] * PolynomialPowerMod[f[ $\tau$  - 1], 2, {f1, q}];
  p5 = p5 - f[ $\tau$  - 2] * PolynomialPowerMod[f[ $\tau$  + 1], 2, {f1, q}];
  p5 = PolynomialPowerMod[p5, q, {f1, q}];
  (*  $p_6 = y^{q^2-1} = (y^2)^{(q-1)(q+1)/2}$  *)
  p6 = PolynomialPowerMod[y2, (q - 1), {f1, q}];
  p6 = PolynomialPowerMod[p6, (q + 1) / 2, {f1, q}];
  Print["{k,  $\tau$ } = ", {k,  $\tau$ }];
  If[EvenQ[k],
    (*  $4^q f_{\tau}^{3q} ((2 x^{q^2} + x) \alpha \beta^2 - \beta^3 y^{q^2-1} - y^2 \alpha^3) y^2 f_k^2 - \alpha \beta^2 f_{k-1} f_{k+1}$  *)
    p19 = PolynomialPowerMod[p3 *  $\alpha$  * b2 - b3 * p6 - y2 * a3, 1, {f1, q}];
    p19 = PolynomialPowerMod[p4 * (p19 * y2 * p2 - p1), 1, {f1, q}];
    If[EvenQ[ $\tau$ ],

```



```

(* 4^q f_c^3 q y^3 q+1 ((2 x^q^2 +x) alpha beta^2 - beta^3 y^q^2-1 - y^2 alpha^3) y^2 f_k^2 -
   alpha beta^2 f_{k-1} f_{k+1} ) - beta^3 y^2 f_k^2 (f_{tau-1}^2 f_{tau+2} - f_{tau-2} f_{tau+1}^2)^q *)
p19 *= PolynomialPowerMod[y2, (3 q + 1) / 2, {f1, q}];
p19 = PolynomialPowerMod[p19 - b3 * y2 * p2 * p5, 1, {f1, q}];
',
(* 4^q f_c^3 q ((2 x^q^2 +x) alpha beta^2 - beta^3 y^q^2-1 - y^2 alpha^3) y^2 f_k^2 -
   alpha beta^2 f_{k-1} f_{k+1} ) - beta^3 f_k^2 (f_{tau-1}^2 f_{tau+2} - f_{tau-2} f_{tau+1}^2)^q y^q+1
*)
yq = PolynomialPowerMod[y2, (q + 1) / 2, {f1, q}];
p19 = PolynomialPowerMod[p19 - b3 * p2 * p5 * yq, 1, {f1, q}];
];
];
If[OddQ[k],
y4 = PolynomialPowerMod[y2, 2, {f1, q}];
(* p7 = beta^3 (y^2)^(q^2+3)/2 = beta^3 (y^2)^2 (y^2)^(q^2-1)/2 *)
p7 = PolynomialPowerMod[b3 * y4 * p6, 1, {f1, q}];
p19 = PolynomialPowerMod[p3 * alpha * b2 * y2 - p7 - a3, 1, {f1, q}];
p19 -= PolynomialPowerMod[p4 * (p19 * p2 - y4 * p1), 1, {f1, q}];
If[EvenQ[tau],
(* 4^q f_c^3 q y^3 q-3 (((2 x^q^2 +x) alpha beta^2 y^2 - beta^3 y^q^2+3 - alpha^3) f_k^2 -
   y^4 alpha beta^2 f_{k-1} f_{k+1} ) - beta^3 f_k^2 (f_{tau-1}^2 f_{tau+2} - f_{tau-2} f_{tau+1}^2)^q *)
p19 *= PolynomialPowerMod[y2, (3 q - 3) / 2, {f1, q}];
p19 = PolynomialPowerMod[p19 - b3 * p2 * p5, 1, {f1, q}];
',
(* 4 f_c^3 q ((2 x^q^2 +x) alpha beta^2 y^2 - beta^3 y^q^2+3 - alpha^3) f_k^2 -
   alpha beta^2 y^4 f_{k-1} f_{k+1} ) - beta^3 f_k^2 (f_{tau-1}^2 f_{tau+2} - f_{tau-2} f_{tau+1}^2)^q y^q+3 *)
(* 4^q f_c^3 q ((x^q^2 +x) alpha beta^2 y^2 - beta^3 y^q^2+3 - alpha^3) f_k^2 -
   alpha y^4 beta^2 f_{k-1} f_{k+1} ) - beta^3 f_k^2 (f_{tau-1}^2 f_{tau+2} - f_{tau-2} f_{tau+1}^2)^q y^q+3 *)
yq = PolynomialPowerMod[y2, (q + 3) / 2, 1, {f1, q}];
p19 = PolynomialPowerMod[p19 - b3 * p2 * p5 * yq, 1, {f1, q}];
];
];
Return[p19];
];

```

■ `ComputeTmodLCaseTwo[ q, l,  $\tau$ , {a, b} ]`

Given that  $\phi_l^2 P \neq \pm k P$  for any  $P \in E[l]$  we must test equation (12) for each  $\tau$  in  $\mathbb{Z}/l\mathbb{Z}^\times$  as  $\phi_l^2 P + qP = \pm \tau \phi_l P$ . These tests are performed using the equations of **Schoof(19)**.

```

ComputeTmodLCaseTwo[q_, l_, {a_, b_}] :=
Module[{f1, k,  $\tau$ , g, p19x, p19y},
  f1 = f[l];
  k = Mod[q, l];
  For[ $\tau$  = 1,  $\tau$  < l, ++ $\tau$ ,
    p19x = ComputeEquation19X[q, l,  $\tau$ , {a, b}];
    Print["p19x = ", p19x];
    g = PolynomialGCD[p19x, f1, Modulus → q];
    Print["gcd(p19x, f1) = ", g];
    (* g ≠ 1 means such P exists, so check the y coordinate *)
    If[! SameQ[g, 1],
      p19y = ComputeEquation19Y[q, l,  $\tau$ , {a, b}];
      Print["p19y = ", p19y];
      g = PolynomialGCD[p19x, f1, Modulus → q];
      Print["gcd(p19y, f1) = ", g];
      If[SameQ[g, 1], Return[1 -  $\tau$ ], Return[ $\tau$ ]];
    ];
  ];
  (* No solution for any  $\tau$ , return 0 *)
  Return[0];
]

```

■ `ComputeGroupOrderSchoof`[ $\{a, b\}, p$ ]

Computes the order of the group of points  $E(\mathbb{F}_p)$  on the elliptic curve  $E: y^2 = x^3 + Ax + B$  over  $\mathbb{F}_p$  using Schoof's Algorithm.

```

ComputeGroupOrderSchoof[{a_, b_}, p_] :=
Module[{s, t, l, fl, tl, prmprod, m,
  i, g, gy, tau, p16, p19x, p19y, r, grpord},
(* Create a set of smallest primes S such that  $\prod_{i=1}^m l_i > 4\sqrt{p}$  *)
{prmprod, s} = ComputePrimeSet[p];
Print["Primeset = ", s];
(* Compute t (mod 2) *)
tl = ComputeTModTwo[{a, b}, p];
Print["tau (mod 2) = ", tl];
t = {{2, tl}};
(* Create the required
  division polynomials  $f_k$ . psi is a global! *)
m = Last[s];
psi = ComputeDivisionPolynomials[{a, b}, p, m];
(* Now for each other  $l \in S$  compute t mod l as follows *)
For[i = 2, i <= Length[s], ++i,
  l = s[[i]];
  fl = f[l];
  Print["f[" , l, "] = ", fl];
  (* If  $\exists P \in E[l] \ni \phi_q^2 P = \pm kP$  we are in case 1 *)
  p16 = ComputeEquation16[p, l, {a, b}];
  g = PolynomialGCD[p16, fl, Modulus -> p];
  Print["p16 = ", p16];
  Print["gcd(p16, f1) = ", g];
  If[! SameQ[g, 1],
    (* Determine t (mod l) using Schoof (17) and (18) *)
    tl = ComputeTmodLCaseOne[p, l, {a, b}];
    AppendTo[t, {l, tl}];
  ,
  (* Else, we are in case 2 testing  $\phi_q^2 P + kP = \pm r\phi_q P$  *)
  (* Determine if  $\exists \tau < l$  satisfying Schoof (19) *)
  For[tau = 1, tau < (l + 1) / 2, ++tau,
    Print["tau1 = ", tau];
    (* If (19 X) holds for some  $x \in E[l]$  *)
    p19x = ComputeEquation19X[p, l, tau, {a, b}];
    Print["p19x = ", p19x];
    g = PolynomialGCD[p19x, fl, Modulus -> p];
    Print["gcd(p19x, f1) = ", g];
    If[! SameQ[g, 1],
      (* then  $\phi_q^2 P + kP = \pm r\phi_q P$ ,
        use (19 y) to determine the sign *)
      p19y = ComputeEquation19Y[p, l, tau, {a, b}];
      Print["p19y = ", p19y];
      gy = PolynomialGCD[p19y, fl, Modulus -> p];

```

```

Print["gcd(p19y, f1) = ", gy];
If[! SameQ[gy, 1], t1 = τ, t1 = 1 - τ];
AppendTo[t, {1, t1}];
(* There can be only one such τ for each l *)
Break[];
];
];
];
Print["τ (mod li) = ", t];
];
(* Convert t from {{1, t (mod l)}} to {{1},{t1}} *)
t = Transpose[t];
Print["τ (mod li) = ", t];
(* Use the Chinese Remainder Theorem to compute the smallest
r > 0 such that r ≡ ti (mod li) for each i *)
r = DetermineChineseRemainder[Last[t], First[t]];
(* force  $-2\sqrt{p} \leq r \leq 2\sqrt{p}$  *)
If[r > 2  $\sqrt{p}$ , r -= prmprod];
Print["τ = ", r];
grpord = p + 1 - r;
Return[grpord];
];

```