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

John McGee
Radford University
14-August-2007

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.

Acknowlegements

This material was developed as part of my Master's Thesis completed at Virginia Tech in June of 2006. I am forever indebted to my advisors Charles Parry for help with number theory, Michael Williams for extensive help with *Mathematica* and expecially to Ezra Brown who inspired this project and supported and guided me through the work. I also thank Lawrence Washington for his excellent text [7] which provided the basis for my work and for his willingness answering questions concerning Schoof's algorithm.

This document represents an update of that work in which errors in my *Mathematica* code have been corrected, testing has been extended, and code restructuring was done to improve performance and provide compatibility with *Mathematica* version 6.0.

Chapter 1 - Introduction

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

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

$$y^2 = x^3 + A x + B. \tag{1}$$

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 $q = p^k$, 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

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

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:

$$E : y^2 = x^3 + A x + B. \tag{3}$$

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

$$y^2 = (x - a)(x - b)(x - c) = x^3 - (a + b + c)\,x^2 + (a\,b + a\,c + b\,c)\,x - a\,b\,c.$$

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 $2\,a + c = 0$ 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 $4\,A^2 + 27\,B^2 = 0$. The contrapositive gives that (3) has distinct roots only if

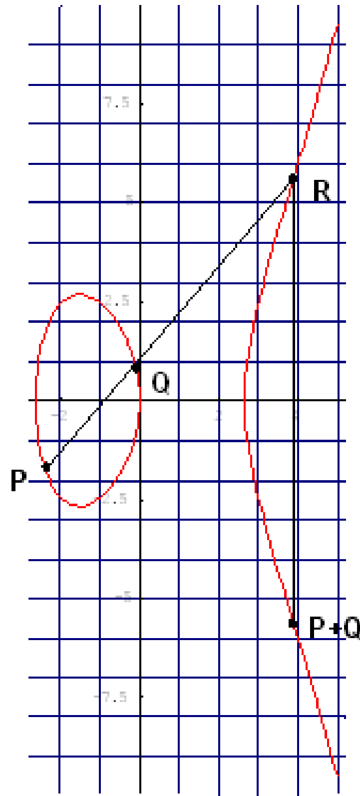
$$4\,A^2 + 27\,B^3 \neq 0. \tag{4}$$

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 $y^2 = x^3 - 5\,x - 2$. 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, $\mathcal{O}$. This definition is justified based on a consideration of the elliptic curve in projective coordinates, where $\mathcal{O}$ is a well-defined point.

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



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 symmetry 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 $E : y^2 = x^3 + A x + B$ with distinct x-coordinates. Then the slope of the line between the two points is given by

$$\lambda = (y_2 - y_1) / (x_2 - x_1). \tag{5}$$

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

$$y - y_1 = \lambda(x - x_1). \tag{6}$$

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

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

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

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

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

$$x^3 - (x_1 + x_2 + x_3)\,x^2 + (x_1\,x_2 + x_1\,x_3 + x_2\,x_3)\,x - x_1\,x_2\,x_3. \tag{8}$$

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

$$x_3 = \lambda^2 - x_1 - x_2. \tag{9}$$

Now the corresponding y-coordinate is obtained using the equation of the line (6) and then by negating the result to obtain the y-coordinate of $P + Q$,

$$y_3 = -(\lambda(x_3 - x_1) + y_1) \;=\; \lambda(x_1 - x_3) - y_1. \tag{10}$$

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 $P + Q = O$, the identity. Otherwise $P = Q$, so we want to compute $P + P = 2\,P$. To accomplish this we define λ 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

$$2\,y\,\frac{d\,y}{d\,x} = 3\,x^2 + A$$

so that, for $y_1 \neq 0$,

$$\lambda = \frac{d\,y}{d\,x} = \frac{3\,x_1{}^2 + A}{2\,y_1}. \tag{11}$$

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

$$(x_1,\, y_1) \oplus (x_1,\, y_1) = \left(\lambda^2 - 2\,x_1,\, \lambda(x_1 - x_3) - y_1\right). \tag{12}$$

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 $\mathbb{F}_p$

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 $E: y^2 = x^3 + A\,x + B$ is an element of $\mathbb{F}_q[x, y]$, which means that $A, B \in \mathbb{F}_q$. Then, if $4\,a^3 + 27\,b^2 \neq 0$ in $\mathbb{F}_q$ it can be shown that

$$E(\mathbb{F}_q) = \left\{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q \,\middle|\, y^2 = x^3 + A\,x + B\right\}$$

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 $2\,q$ 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 $5\,c \equiv 1 \pmod 7$, 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

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

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 $s_i = 0$ of p_i does not divide b. Then the greatest common divisor d is given by

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

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

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

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

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

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

$$r_0 = s_2 r_1 + r_2$$
$$\ldots$$
$$r_{i-1} = s_{i+1} r_i + r_{i+1}.$$

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

Note, however, that if $x = q\, y + r$ 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

$$\gcd(y, r) = u\, y + v\, r = u\, y + v(x - q\, y) = (u - v\, q)\, y + v\, x,$$

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 $d = r\,a + s\,b$. 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 ExtendedEucideanAlgorithm[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, lets work over $\mathbb{F}_{19}$, the field with 19 elements {0, 1, ..., 18} with arithmetic modulo 19, a prime. In a field, every nonzero element has a multiplicative inverse, so lets 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 kth bit of n is 1 then $x \equiv x * a^k \pmod{p}$.
Then $a^{2k} \equiv a^k * a^k \pmod{p}$ 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 $n\,m$ 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
$11^{37} = 340\,039\,485\,861\,577\,398\,992\,406\,882\,305\,761\,986\,971$ in $\mathbb{Z}$ and then find
$11^{37} = 3\,505\,561\,709\,913\,169\,061\,777\,390\,539\,234\,659\,659 * 97 + 48$ so that $11^{37} \equiv 48 \pmod{97}$.
However, in binary 37 = "100101"$_{\text{"2"}}$, so we can compute $11^{37} \pmod{97}$ as follows.

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

But $11^2 \equiv 24 \pmod{97}$, $24^2 \equiv 91 \pmod{97}$ and $91^2 \equiv 36 \pmod{97}$ so that
$11^{37} \equiv \left((36 * 11)^2\right)^2 * 11 \pmod{97}$. Then $396 \equiv 8 \pmod{97}$, $\left(8^2\right)^2 \equiv 22 \pmod{97}$,
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 $w^2 \equiv p \pmod{l}$ for w, with l a prime number less than $\sqrt{p}$. In other words we need to find the square root of p modulo l. Unlike the multiplicative inverse problem in a field, this problem does not always have a solution. When it does, we say that p is a quadratic residue modulo l, else p 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

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

Otherwise, for all $i < p$, $\gcd(i, p) = 1$. Then for each i less than p, $i\,j \equiv a \pmod{p}$ has a unique solution, which can not be $j = i$, else $i^2 \equiv a (\text{mod } p)$. 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

$$p = 360\,027\,784\,083\,079\,948\,259\,017\,962\,255\,826\,129\,.$$

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

$$x = 203\,744\,876\,602\,447\,660\,339\,212\,047\,901\,408\,164\,.$$

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 $z < N$

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

where $N = n_1 n_2 \ldots n_k$ 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

$$a_i = \{385, 154, 110, 70\}$$
$$b_i = a_i^{-1} (\text{mod } n_i) = \{1, 4, 3, 3\}$$
$$e_i = a_i\, b_i = \{385, 616, 330, 210\}$$
$$z = e \cdot r = 3835 \equiv 755 \pmod{N}$$

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 $k\,P$, 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 6 - 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 $E : y^2 = x^3 + 46\,x + 74$ over $\mathbb{F}_{97}$. It has discriminant

$$d \equiv -\left(4\,a^3 + 27\,b^2\right) \equiv -\left(4 * 46^3 + 27 * 74^2\right) \equiv 537\,196 \equiv 87 \pmod p$$

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

At $x = 1$ we have

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

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 $P + P = 2\,P$ using Equation (12) modulo p as

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

Then

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

Hence $2 * (1, 11) = (22, 26)$ on $E(\mathbb{F}_{97})$. We can also compute $4 * (1, 11)$ by adding $(22, 26) + (22, 26)$ 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 $k\,P = \mathcal{O}$. One way to determine k is to compute $n\,P$ for each successive n starting at 1 until the result is the identity. For the previous example with $E : y^2 = x^3 + 46\,x + 74$ over $\mathbb{F}_{97}$ and $P = (1, 11)$ we find, using repeated application of **EcAddMod**, that

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

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 $z = x^3 + A x + B$ 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 $2 q + 1$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 $E : y^2 = x^3 + 46 x + 74$ 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 $z = x^3 + A x + B$ 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 $\mathcal{O}$, 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 of points for $y^2 = x^3 + 46 x + 74$ over $\mathbb{F}_{97}$

x	y_1	y_2	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

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

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

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

Let $\phi_q : E(\overline{\mathbb{F}}_q) \to E(\overline{\mathbb{F}}_q)$ 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 ϕ_q is an endomorphism called the Frobenius map. It has the following property, crucial to Schoof's algorithm.

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

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

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

The Chinese Remainder Theorem is then applied to compute the unique

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

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 sections 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

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

Sketch of the Proof

Define the map $(\phi_q - 1) : E(\overline{\mathbb{F}}_q) \to E(\overline{\mathbb{F}}_q)$, 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 ϕ_q is the identity on $E(\mathbb{F}_q)$. Further, since

$\phi_q - 1$ is a separable polynomial then

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

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

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

So we can conclude that

$$\deg(r\,\phi_q - s) = r^2\,q + s^2 - r\,s\,t. \tag{17}$$

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

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

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

$$q\,x^2 - \tau\,x + 1 \geq 0. \tag{19}$$

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

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

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 $\sharp E(\mathbb{F}_p)$, then we can compute $\sharp E(\mathbb{F}_{p^n})$ in a direct manner.

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

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

So we only need to use Schoof's Algorithm to solve for $\sharp E(\mathbb{F}_p)$. Then we can compute $\sharp E(\mathbb{F}_q) = \sharp E(\mathbb{F}_{p^n})$ via (20). Of course, if p is a small prime, then it is easy to determine $\sharp E(\mathbb{F}_p)$ 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 α and β. The following recursion relation computes $s_n = (\alpha^n + \beta^n)$ where

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

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

For our example elliptic curve $E: y^2 = x^3 + 46\,x + 74$ 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 $\# E(\mathbb{F}_{p^4}) = 88\,531\,200$.

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 $k\, P = \mathcal{O}$ 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 7 - Determining Group Order using Hasse's Theorem

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

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

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 $79 \le 80 \le 117$ 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 $E: y^2 = x^3 + A x + B$, 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 $E: y^2 = x^3 + A x + B$. 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 $x^3 + A x + B = 0$.

Suppose that there exists some $e \in \mathbb{F}_p$ such that $e^3 + A e + B = 0$ 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 $2 * \left(\frac{n}{2} * 1\right) = n \equiv 0 \pmod n$, 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 $2 * \left(\frac{n}{2} * P_g\right) = n P_g = \infty$, 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

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

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 $2 P_2 = \infty$.

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 $x^3 + A x + B$ has a root in $\mathbb{F}_p$.

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

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

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 in encoded in the *Mathematica* function **ComputeTModTwo**.

Example 8 - 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 $E: y^2 = x^3 + a x + b$. 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 $E: y^2 = x^3 + a x + b$, that is, the set of points in $E(\overline{\mathbb{F}}_p)$ with order dividing n, so that $E[n] = \{P \in E(\overline{\mathbb{F}}_p) \mid n P = \mathcal{O}\}$. 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 ψ_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.

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

Lets see why ψ_3 is the correct polynomial. First, if $P = (x, y) \in E[3]$ then $3 P = 0$ which means that $2 P = -P$, hence the x-coordinates of $2 P$ and $-P$ must be the same. Using Equations (11,12) to compute $2 P$ we find

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

so that

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

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

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

So if $\psi_3 = 0$ then $2 P = \pm P$, meaning that $P = \mathcal{O}$ 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 $x^3 + A x + B$. More generally we can replace y^{2k} with $(x^3 + A x + B)^k$. 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?

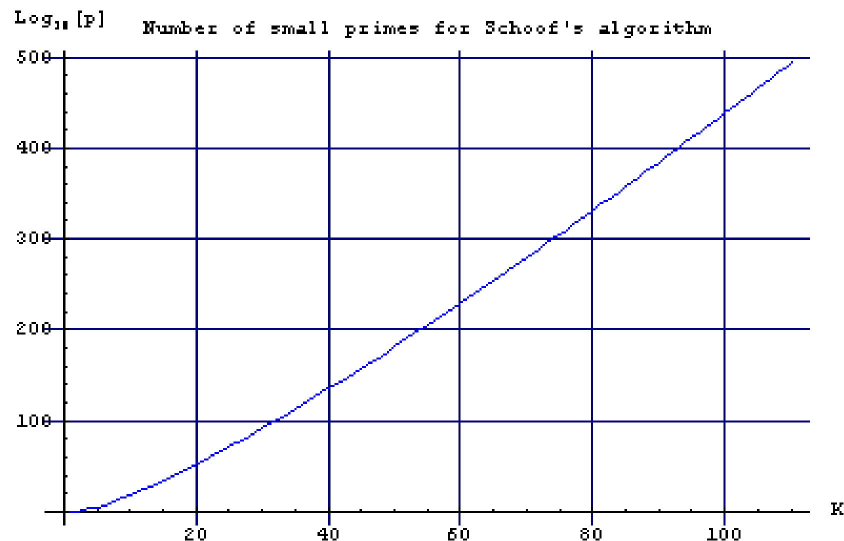
How many division polynomials will we need for the execution of Schoof's algorithm? As noted in the outline of Schoof's algorithm in chapter 4, we need to test Equation (15) for a set of primes l_i such the product of these primes is greater than $4\sqrt{p}$. The function **ComputePrimeSet** determines this set. If the cardinality of $\{l_i\} = k$, then ψ_{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 $k > 10$.

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

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 4 - Number of digits in *p* vs. number of small primes.



Example 9 - 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 ψ_9, so we compute these at this time. Note that the odd numbered polynomials, such as $\psi_1, \psi_3, \ldots$ 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

$$\psi_1(x, y) = 1,$$
$$\psi_2(x, y) = 2y,$$
$$\psi_3(x, y) = 18 + 15x + 82x^2 + 3x^4, \ \psi_4(x, y) = \left(61 + 50x + 69x^2 + 3x^3 + 47x^4 + 4x^6\right)y,$$
$$\psi_5(x, y) = 23 + 67x + 11x^2 + 38x^3 + 77x^4 + 43x^5 + 93x^6 + 26x^7 + 47x^8 + 87x^9 + 39x^{10} + 5x^{12}.$$

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 $2 \le n \le 8$ giving

$$f_2[4] = 2, \ f_3[4] = 24, \ f_4[4] = 0, \ f_5[4] = 47,$$
$$f_6[4] = 25, \ f_7[4] = 22, \ f_8[4] = 0,$$

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 *n P* with the Division Polynomials

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

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

It can be shown that multiplication by n is an endomorphism μ_n of $E(\overline{\mathbb{F}}_p)$. This follows from the fact that $E(\overline{\mathbb{F}}_p)$ is an abelian group so that $n(P + Q) = n\,P + n\,Q$. Since $n\,P = \mathcal{O}$ if and only if $P \in E[n]$ we have that the kernel of μ_n is $E[n]$. Further, because μ_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 $n\,P$ 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 $\phi_q : E(\overline{\mathbb{F}}_q) \to E(\overline{\mathbb{F}}_q)$ 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 $\phi_q(x_1, y_1) = \left(x_1^q, y_1^q\right)$. Substituting this into the elliptic curve equation gives

$$\left(y_1^q\right)^2 = \left(y_1^2\right)^q = \left(x_1^3 + A\,x_1 + B\right)^q.$$

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

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

Hence $\left(x_1^q, y_1^q\right) = \phi_q(x_1, y_1) \in E(\overline{\mathbb{F}}_q)$, so that ϕ_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

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

Using the same properties of qth powers in $\overline{\mathbb{F}}_q$ we have

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

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

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 ϕ_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 $E(\overline{\mathbb{F}}_q)$ who's order divides l. First we show that $E[l]$ is a subgroup of $E(\overline{\mathbb{F}}_q)$. Clearly $\mathcal{O} \in E[l]$. If $P, Q \in E[l]$ then, because $E(\overline{\mathbb{F}}_q)$ is abelian, $\mathcal{O} = l\,P + l\,Q = l(P+Q)$, so that $P+Q \in E[l]$. Further,

$$\mathcal{O} = l\,\mathcal{O} = l(P + (-P)) = l\,P + l(-P) = \mathcal{O} + l(-P) = l(-P),$$

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 α 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

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

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

$$\begin{aligned}&\deg(\phi_q - 1) \equiv \det(\phi_{q,l} - I)\,(\mathrm{mod}\,l)\\ &= (s-1)(v-1) - t\,u = s\,v - t\,u - (s+v) + 1.\end{aligned}$$

But $s\,v - u\,t = \det(\phi_{q,l}) \equiv q\,(\mathrm{mod}\,l)$ (by Washington [7] Proposition 3.15).

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

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

Hence we have the following congruences. First, $a \equiv (s+v)\,(\mathrm{mod}\,l)$, the trace of $\phi_{q,l}$. Also $t\,u \equiv s\,v - q\,(\mathrm{mod}\,l)$.

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

$$\begin{pmatrix} s & t \\ u & v \end{pmatrix}^2 - a\begin{pmatrix} s & t \\ u & v \end{pmatrix} + q\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} s^2 + t\,u & s\,t + t\,v \\ s\,u + u\,v & t\,u + v^2 \end{pmatrix} - \begin{pmatrix} a\,s - q & a\,t \\ a\,y & a\,v - q \end{pmatrix}$$

$$= \begin{pmatrix} s^2 + t\,u - a\,s + q & s\,t + t\,v - a\,t \\ s\,u + u\,v - a\,u & v^2 + t\,u - a\,v + q \end{pmatrix}.$$

Applying the congruences yield

$$\equiv \begin{pmatrix} s^2 + s\,v - q - (s + v)\,s + q & (s + v)\,t - (s + v)\,t \\ (s + v)\,u - (s + v)\,u & v^2 + s\,v - q - (s + v)\,v + q \end{pmatrix}.$$

$$\equiv \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} (\bmod\ l).$$

Therefore $\left(\phi_{q,l}\right)^2 - a\left(\phi_{q,l}\right) + q \equiv 0\ (\bmod\ l)$ 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,

$$\phi_q^2 + k \equiv \tau\,\phi_q\ (\bmod\ l) \text{ where } k \equiv q\ (\bmod\ l) \text{ and } \tau \equiv a\ (\bmod\ l).$$

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

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

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 ψ_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

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

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

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

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

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

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 τ.

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 ψ_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

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

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

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:

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

For w odd:

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

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 $t \equiv -2\,w \pmod{l}$, else such a point exists and $t \equiv 2\,w \pmod{l}$. 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

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

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

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

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

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

For k even we have

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

and

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

Otherwise, for k odd

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

and

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

We use these equations for α and β 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

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

then x_3 is given by

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

Also y_3 is given by

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

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

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

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

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

Expanding and clearing the (nonzero) denominators gives

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

Bringing everything to the left hand side we have

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

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 ψ_l.

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

5.13 Schoof Equation (19y)

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

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

we must test the y-coordinate to determine the sign of τ. 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 $\beta^3\,\psi_k^2\,4\,y^p\,\psi_t^{3\,p}$ to clear the denominators we have

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

Rearranging gives for Schoof (19y) (corrected)

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

For k even, τ even, we must take $\alpha \to y\,\alpha$ so that,

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

For k even, τ odd,

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

For k odd, τ even we must take $\beta \to y\,\beta$ so that,

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

For k odd, τ odd,

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

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

5.14 Schoof's Algorithm Summary

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

1. If $\gcd(x^3 + a\,x + b,\ x^p - x) = 1$ 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 ψ_k.
4. For each $l \in S$, compute $k \equiv p(\bmod\ l)$
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 $t \equiv 2\,w \pmod l$, else $t \equiv -2\,w \pmod l$.
10. else we are in case two
11. For each $\tau \le (l+1)/2$
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 τ
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 $E : y^2 = x^3 + 46\,x + 74$ 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

$x^p(\bmod x^3 + a\,x + b) = 47 + 60\,x + 30\,x^2$ and
$\gcd(x^p - x,\ x^3 + a\,x + b) = 40 + x$

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 $k \equiv p(\bmod 5)$ and we find

"f["5"] ="23 + 67 x + 11 x^2 + 38 x^3 + 77 x^4 + 43 x^5
+ 93 x^6 + 26 x^7 + 47 x^8 + 87 x^9 + 39 x^{10} + 5 x^{12}

"\!\(p_16\) = "7 + 91 x + 40 x^2 + 24 x^3 + 81 x^4 + 69 x^5
+ 43 x^6 + 45 x^7 + 39 x^8 + 14 x^9 + 30 x^{10} + 79 x^{11}

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

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

$p_{19\,x} \equiv 0\,(\bmod f_l,\ p)$ and

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

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

"p19y ="39 + 52 x + 48 x^2 + 33 x^3 + 91 x^4 + 3 x^5
+ 23 x^6 + 59 x^7 + 16 x^8 + 37 x^9 + 33 x^{10} + 74 x^{11}

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

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 conguences.

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

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 $\sharp E(\mathbb{F}_p) = 80$.

6.2 Other Experiments

Table 1 summarizes the results of running Schoof's algorithms on several curves. The first column describes the curve $y^2 = x^3 + A\,x + B$ 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 Hasses' bounds that satisfies the congruences of column two. $\sharp E(\mathbb{F}_p)$ is then computed as $p + 1 - t$.

Table 1 - Results of Schoof's Algorithm

A, B, p	t (mod l_i)	t	$\sharp E(\mathbb{F}_p)$	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}, {2, 11}, {0, 13}	-13	3559	yes

Several larger problems have been run, the output of the largest of these runs follows.

```
r = Timing[EcGroupOrderSchoof[ec13]]
E(𝔽2184972043155134786254507044209):
 x^3 + 812066892826496630200758496181 x + 393895671555005480312164618491
Hasse's bounds 2184972043155131829924320373490
  ≤ ♯E(𝔽p) ≤ 2184972043155137742584693714930
⌈4√p⌉ = 5912660373341444
∏ li = 32362620136236390
{2, 3, 5, 7, 11, 13, 17, 23, 29, 31, 37, 41, 43, 47}
Computing 49 division polynomials
τ (mod 3 ) = {{2, 0}, {3, 0}}
{k,τ} = {4, 2}
τ (mod 5 ) = {{2, 0}, {3, 0}, {5, 2}}
```

```
{k,τ} = {5, 1}
τ (mod 7 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}}
τ (mod 11 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10}}
{k,τ} = {9, 1}
τ (mod 13 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10}, {13, 12}}
{k,τ} = {6, 7}
τ (mod 17 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10}, {13, 12}, {17, 10}}
{k,τ} = {15, 8}
τ (mod 23 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10}, {13, 12}, {17, 10}, {23, 8}}
{k,τ} = {27, 12}
τ (mod 29 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10}, {13, 12}, {17, 10}, {23, 8}, {29, 17}}
{k,τ} = {28, 13}
τ (mod 31 ) =
 {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10}, {13, 12}, {17, 10}, {23, 8}, {29, 17}, {31, 18}}
{k,τ} = {34, 4}
τ (mod 37 ) =
 {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10}, {13, 12}, {17, 10}, {23, 8}, {29, 17}, {31, 18}, {37, 4}}
{k,τ} = {19, 2}
τ (mod 41 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10},
  {13, 12}, {17, 10}, {23, 8}, {29, 17}, {31, 18}, {37, 4}, {41, 2}}
{k,τ} = {19, 18}
τ (mod 43 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10},
  {13, 12}, {17, 10}, {23, 8}, {29, 17}, {31, 18}, {37, 4}, {41, 2}, {43, 18}}
{k,τ} = {43, 17}
τ (mod 47 ) = {{2, 0}, {3, 0}, {5, 2}, {7, 1}, {11, 10}, {13, 12},
  {17, 10}, {23, 8}, {29, 17}, {31, 18}, {37, 4}, {41, 2}, {43, 18}, {47, 17}}
τ = -2 235 306 593 865 918
{44 959.2, 2 184 972 043 155 137 021 561 100 910 128 }
```

The previous result $\# E(\mathbb{F}_p) = 2\,184\,972\,043\,155\,137\,021\,561\,100\,910\,128$ was produced in a runtime of approximately 12.5 hours on a T2500 equipped laptop running at 2 GHz with 1 GB of memory.

6.3 Discussion of Results

Schoof's method, as here implemented, has primarily 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 55th division polynomial, which produces intermediate products of degree greater than 9×10^6. The improvements due to Elkies and Atkin, called the SEA algorithm, reduce to nearly linear growth the degree of certain divisors of the division polynomials which we can use in their place, making the method applicable to elliptic curves of with cryptographic utility. The software bug and performance issues referred to in an earlier version of this document has been corrected. Our implementation is believed to be correct for all cases based on cross-testing against an algorithm employing the Baby-Step, Giant-Step method to determine random EC point orders. Performance was enhanced by replacing PolynomialMod[P, Q, p] with PolynomialRemainder[P, Q, x, Modulus → p] and by significant code factoring to reduce redundant computations.

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 authors plans to make this suite of elliptic curve functions available as a *Mathematica* package.

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 Téorie des Nombres de Bordeaux 7 (1995), 219-254

[7] Washington, Lawrence - *Elliptic Curves - Number Theory and Cryptography,* Chapman & Hall (2003)

Appendix - Mathematica Code

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];
  ];
```

ExtendedEucideanAlgorithm[*a*, *b*]

Computes $d = \gcd(a, b)$ and returns $\{d, r, s\}$ such that $d = r\,a + s\,b$. The results are returned as the list $\{d, r, s\}$. This code is based on Rosen [4] §3.3.

```
ExtendedEucideanAlgorithm[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 $\gcd(a, p) = 1$, 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} = ExtendedEucideanAlgorithm[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 k^th bit is 1 set x ≡ x*a^k(mod p) *)
    If[Mod[b, 2] ≠ 0,
     x = Mod[x * t, p];
    ];
    (* a^(2^k) ≡ (a^(2^(k-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, 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_] := PowerMod[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 nonresidues, this will occur within 5 trials with better than 95% probability.

```
RandomNonResidue[p_] :=
 Module[{x}, x = 3; While[QuadraticResidueQ[x, p],
   x = RandomInteger[{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 {}. Note: We could use PowerModList[a, 1/2, p] in *Mathematica* 6.0, but it does not appear to be any faster.

```
SqrtModPShanksTonelli[a_, p_] := Module[{s, e, n, q, i, 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 ∋ 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 ∋ 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 ∋ b^(2^m) ≡ 1 *)
  y = b; m = 0;
  While[y ≠ 1, y = Mod[y * y, p]; ++m;];
  If[m == 0, Return[x]];
  (* Compute y = g^(2^(r-m-1)) *)
  y = g;
  For[i = 1, i ≤ r - m - 1, ++i; y = Mod[y * y, p]];
  (* x = x*g^(2^(r-m-1)) *)
  x = Mod[x * y, p];
  (* b = b*g^(2^(r-m)) *)
  y = Mod[y * y, p];
  b = Mod[b * y, p];
  (* g = g^(2^(r-m)) , r = m *)
  g = y; r = m;
 ];
];
```

DetermineChineseRemainder[*t*]

Solve the set of integer congruences

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

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

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

```
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 = e1r1+e2r2+...+ekrk *)
  z = e.r;
  Print["z = e.r = ", z];
  Return[Mod[z, m]];
 ]
```

Modular Polynomial Arithmetic

PolynomialPowerMod[p, n, {q, k}]

Compute $p^n \bmod (q, k)$ where p, q are polynomials, n, k are positive integers

Efficient method using squares of $p \bmod (q, k)$, for example

$p^8 = ((p^2)^2)^2$ where we can reduce mod(q, k) at each step

$p^{11} = p^8 * p^2 * p$, where we can reduce mod (q, k) at each step

Example:

```
PolynomialPowerMod[(x^3 + 2 x + 3), (p + 1) / 2, {x^101 + x^2 + 31, p}]
```

runs in less than 1 second for $p = 32\,452\,867$

```
PolynomialPowerMod[p_, n_, {q_, k_}] := Module[{i, r, s},
  i = n; r = 1; s = p;
  While[i ≠ 0,
   If[EvenQ[i],
     i = Quotient[i, 2];
     s = PolynomialRemainder[s^2, q, x, Modulus → k];
     ,
     i--;
     r = PolynomialRemainder[r * s, q, x, Modulus → k];
    ];
  ];
  Return[r];
 ]
```

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

Compute the first k division polynomials for the elliptic curve $E : y^2 = x^3 + A\,x + B$ 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_] := ψ[[n + 2]] /. y → 1;
```

Elliptic Curve Algorithms (mod p)

■ 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 $y^2 = x^3 + a\,x + b$ over $\mathbb{F}_p$ this is given by

$$d \equiv -\left(4\,a^3 + 27\,b^2\right) (\text{mod } p)$$

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

IsEcQ[{a, b, p}]

$E: y^2 = x^3 + a\,x + b$ 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}] ≠ 0;
```

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

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

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

EcJinvariant[{a, b}, p]

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

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

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 4a^3+27b^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 4a^3/(4 a^3+27 b^2) *)
   j = Mod[1728 * 4 * a3 * di, p];
   Return[j];
  ];
```

GenerateRandomPointEC[{a, b, p}]

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

$y^2 = x^3 + A\,x + B$ with all arithmetic modulo p.

```
GenerateRandomPointEC[{a_, b_, p_}] :=
  Module[{x, e1, x1, y1, y2}, e1 = x^3 + a x + b; y2 = 1; While[True,
    x1 = RandomInteger[{1, p - 1}];
    y2 = Mod[e1 /. x → x1, p];
    If[QuadraticResidueQ[y2, 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 $y^2 = x^3 + A x + B$ with $4 A^2 + 27 B^3 \neq 0$

Returns {A,B}

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

EcAddMod[{*a*, *b*, *p*}, P_1, P_2]

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 $\mathcal{O}$. It returns { } if either point does not lie on the elliptic curve.

```
EcAddMod[{a_, b_, p_}, p1_, p2_] := 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, p}, {x1, y1}], Return[{}]];
  If[! IsEcPointQ[{a, b, p}, {x2, y2}], 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 = PowerMod[2 y1, -1, p];
   m = Mod[(3 x1^2 + a) * w, p];
   ,
   (* else compute the slope of the chord *)
   w = PowerMod[x2 - x1, -1, 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*, *p*}, *Q*, *k*]

Compute $k\,Q$ in the abelian group of points on the elliptic curve $E: y^2 = x^3 + a\,x + b$ 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_, p_}, q_, k_] := Module[{i, r, s},
   i = k; r = {∞, ∞}; s = q;
   While[i ≠ 0,
    If[EvenQ[i],
      i = Quotient[i, 2];
      s = EcAddMod[{a, b, p}, s, s];
      ,
      i = i - 1;
      r = EcAddMod[{a, b, p}, r, s];
     ];
   ];
   Return[r];
  ];
```

Elliptic Curve Point and Group Order Methods (mod p)

EcPointOrderMod[ec, *Q*, *m*]

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

$k \in \mathbb{Z}_p \ni k\,Q = \mathcal{O}$ 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[{a_, b_, p_}, p1_, m_] := Module[{k, p2, pn3},
  (* Test for special case orders 1,2 *)
  If[p1 == {∞, ∞}, Return[1]];
  p2 = EcAddMod[{a, b, p}, p1, p1];
  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[{a, b, p}, p2, p1];
   ++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 $E : y^2 = x^3 + A x + B$.

```
FindEcPointSet[{a_, b_, p_}] := Module[{s, x, e1, x1, y1, y2},
   e1 = x^3 + 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 ∋ y1^2 ≡ 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 y^2 = x^3+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 an put it in the last column *)
  t = Map[Append[#, EcPointOrderMod[{a, b, p}, Take[#, 2], 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];
 ]
```

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

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

```
EcPointOrderBabyGiant[{a_, b_, p_}, P1_] :=
  Module[{ec1, Q1, i, j, k, m, b1, t1, t2, p2, pm3, pm4, m1, f1, x1, q2,
    q3, pn5, pn6, pn7},
   ec1 = {a, b, p};
   (* Check for small order directly *)
   k = EcPointOrderMod[ec1, P1, Min[100, 2*p]];
   If[k > 0, Return[k]];
   (* Compute Q = (p+1)P *)
   Q1 = EcPowerMod[ec1, P1, p + 1];
   (* Table size m = ⌈p^(1/4)⌉ *)
   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[ec1, p2, P1];
    AppendTo[t1, p2];
   ];
   t2 = First[Transpose[t1]];
   q2 = Q1;
   q3 = Q1;
   (* Create table of Q + 2kmP for k=-m,...,m*)
   pm3 = EcPowerMod[ec1, P1, 2*m];
   pm4 = pm3;
   (* Compute -P(mod p) *)
   pn5 = Mod[P1*{1, -1}, p];
   pn6 = EcPowerMod[ec1, pn5, 2*m];
   pn7 = pn6;
   k = 0;
   (* Compute for k=±1,±2,... *)
   While[True,
    (* Until Q + 2kmP = ±jP or ... *)
    If[MemberQ[t2, First[q2]],
     Break[];
    ];
    (* Until Q - 2kmP = ±jP or ... *)
    If[MemberQ[t2, First[q3]],
     k = -k;
     q2 = q3;
     Break[];
    ];
```

```
 (* Compute next Q ± 2kmP for next k *)
 q2 = EcAddMod[ec1, Q1, pm4];
 q3 = EcAddMod[ec1, Q1, pn7];
 pm4 = EcAddMod[ec1, pm4, pm3];
 pn7 = EcAddMod[ec1, pn7, pn6];
 ++k;
];
(* Here we have Q ± 2kmP = ±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];
b1 = True;
While[b1,
 b1 = 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[ec1, P1, x1 - 1];
  If[pm3 == pn5,
   m1 = x1;
   b1 = True;
   If[--f1[[i, 2]] ≤ 0, f1 = Delete[f1, i]];
   Break[];
  ];
 ];
];
Return[m1];
];
```

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[{a_, b_, p_}] :=
  Module[{ec1, pr2, h0, h1, p1, p2, k1, k2, or1, or3, t, k},
   ec1 = {a, b, p};
   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[ec1];
    Print["P = ", p1];
    (* Determine |p1| *)
    k1 = EcPointOrderBabyGiant[ec1, p1];
    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√p ≤ LCM ≤ p + 1 +2√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[{}];
  ];
```

Methods to solve the Discrete Log problem

This collection of *Mathematica* functions attack the elliptic curve discrete log problem using the Pollard Rho method.

MakeMList[ec, P1, Q1, ab, *p*]

Make a list of $M_i = a_i P + b_i Q$ on the elliptic curve $E(\mathbb{F}_p)$

where ec $= \{A, B\} \ni E : y^2 = x^3 + A x + B$ and

ab $= \{a_i, b_i\}$

```
MakeMList[ec_, P1_, Q1_, ab_, p_] := Module[{m},
   m = Map[{EcAddMod[ec,
        EcPowerMod[ec, P1, First[#], p],
        EcPowerMod[ec, Q1, Last[#], p],
        p], #} &, ab];
   Return[m];
  ];
```

$s[x] = M_i$ where $i \equiv x \pmod s$ and $s = \# \{M_i\}$

```
s[x_] := m[[Mod[x, Length[m]] + 1]];
```

f[{{x1, y1}, {a1, b1}} ec, *p*] - a function for Pollard's Rho Method

$f[(x, y)] = (x, y) + M_i$ - a function iterated by PollardRhoEcMod

where $(x, y) = a P + b Q$

Return $\{(x', y'), (a', b')\}$, where$(x', y') = (x, y) + M_i = a' P + b' Q$

```
f[{{x1_, y1_}, {a1_, b1_}}, ec_, p_] := Module[{x2, y2, a2, b2},
   {{x2, y2}, {a2, b2}} = s[x1];
   {x2, y2} = EcAddMod[ec, {x1, y1}, {x2, y2}, p];
   {a2, b2} = Mod[{a1, b1} + {a2, b2}, p];
   Return[{{x2, y2}, {a2, b2}}];
  ];
```

PollardRhoEcMod[ec, PX1, QX1, {a0, b0}, *p*]

Implement the Pollard ρ algorithm to solve the discrete log problem on $E(\mathbb{F}_p)$

that is, given $P, Q \in E(\mathbb{F}_p)$, $p \in$ Primes, find $k \ni k P = Q$

```
PollardRhoEcMod[ec_, PX1_, QX1_, {a0_, b0_}, p_] :=
  Module[{p0, p1, p2, i, o, d, k},
   (* Compute the order of P *)
   o = EcPointOrderBabyGiant[ec, PX1, p];
   Print["Order of ", PX1, " = ", o];
   (* P0 = a0*P + b0*Q *)
   p0 = {EcAddMod[ec, EcPowerMod[ec, PX1, a0, p], EcPowerMod[ec, QX1, b0, p],
      p], {a0, b0}};
   (* Compute P1 = f(P0) *)
   p1 = f[p0, ec, p];
   (* Compute P2 = f(P1) *)
   p2 = f[p1, ec, p];
   i = 1;
   (* Until Pi = P2i *)
   While[First[p1] ≠ First[p2] && i ≤ p,
    (* Compute Pi+1 = f(Pi) *)
    p1 = f[p1, ec, p];
    (* Compute P2(i+1) = f(f(P2i) *)
    p2 = f[p2, ec, p];
    p2 = f[p2, ec, p];
    ++i;
   ];
   (* Here with uiP+viQ = u2iP+v2iQ *)
   Print[p1[[2, 1]], " P + ", p1[[2, 2]], " Q = ", p1[[1]]];
   Print[p2[[2, 1]], " P + ", p2[[2, 2]], " Q = ", p2[[1]]];
   (* Then (ui-u2i)P +(vi-v2i)Q = ∞ *)
   d = Mod[Last[p2] - Last[p1], p];
   Print["So that ∞ = ", d[[1]], " P + ", d[[2]], " Q"];
   (* k ≡ (v2i-vi)^-1(ui-u2i) (mod p) *)
   k = Mod[First[d] * MultiplicativeInverse[-Last[d], o], o];
   Print["Then ", -Last[d], "^-1* ", First[d], " ≡ ", k, " (mod ", o, ")"];
   Print["Therefore Q = ", k, " P"];
   Return[k];
  ];
```

Methods for Elliptic Curve Factoring

■ GenerateECFactorBase[*b*]

Generate a factor base with bound b

$$\text{FB} = \left\{p_i^{e_i} \mid p_i \in \mathbb{P}\text{rimes},\ p_i^{e_i} \le b,\ p_i^{e_i+1} > b\right\}$$

```
GenerateECFactorBase[b_] := Module[{f, p, e, m, m1},
  p = 2;
  f = Reap[
    While[p < b,
      m = p; m1 = m;
      While[m < b, m1 = m; m *= p];
      Sow[m1];
      p = NextPrime[p];
     ];
   ];
  Return[Flatten[Last[f]]];
 ]
```

ListProdMod[*t*, *n*]

Compute $\prod_{i=1}^{k} t_i \pmod{n}$

Compute the modulus for each partial product to keep the numbers small

```
ListProdMod[t_, n_] := Module[{p, k},
  p = 1;
  k = Length[t];
  For[i = 1, i ≤ k, ++i,
   p *= t[[i]];
   p = Mod[p, n];
  ];
  Return[p];
 ]
```

LenstraECMFactor[*n*, *t*]

Apply Lenstra's ECM factor method to find a factor of *n* by attempting to compute the multiples of a random point on a random elliptic curve. This has a high probability of failure if *n* is composite, since we are likely to find an integer coordinate that has no inverse mod *n*, and hence is a non-trivial factor of n. Try to factor *n* using up to *t* different curves, sequentially. Returns a factor of *n* if successful, otherwise returns 0.

```
LenstraECMFactor[n_, t_] :=
 Module[{b, f, i, Null, km, q, a, x, y, p1, ec1, pm},
  b = Floor[e^(0.5` Sqrt[Log[n] Log[0.5` Log[n]]])];
  f = GenerateECFactorBase[b]; f = Sort[f];
  For[i = 1, i ≤ t, ++i,
   p1 = RandomInteger[{-b, b}, 2];
   {x, y} = p1;
   a = RandomInteger[{-b, b}];
   ec1 = {a, Mod[y^2 - x^3 - a x, n]};
   k = 1; pm = p1;
   For[i = 1, i ≤ Length[f], ++i,
    h = f[[i]];
    While[h > k, pm = Catch[EcPowerMod[ec1, pm, h, n]];
     If[! NumberQ[First[pm]],
      Print[
       "E: \!\(\*SuperscriptBox[\(y\), \(2\)]\) =
         \!\(\*SuperscriptBox[\(x\), \(3\)]\) + ", First[ec1],
       " x + ", Last[ec1]];
      Print["Factor ", Last[pm], " of ", n, " found while computing ",
       f[[i]], "!*P "];
      Print["on the ", i, "'th try"];
      Return[Last[pm]]; --h;]; k = f[[i]]
     ;]
    ;];
  ];
  Return[0];
 ]
```

Schoof's Algorithm Methods

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

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

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

Note: We may have $p_{16} \equiv 0\ (\mathrm{mod}\ f_l,\ p)$ in which case f_1 divides p_{16} so $\gcd(p_{16},\ f_l) = f_l$

```
ComputeEquation16[{a_, b_, p_}, l_] := Module[{k, fl, y2, p1, p2, p3, p16x},
   fl = f[l];
   k = Mod[p, l];
   y2 = x^3 + a x + b;
   (* p1 = x^(q^2) (mod f1) *)
   p1 = PolynomialPowerMod[x, p, {fl, p}];
   p1 = PolynomialPowerMod[p1, p, {fl, p}];
   (* p2 = (x^(q^2) -x) fk^2(x) (mod f1) *)
   p2 = PolynomialRemainder[f[k]^2, fl, x, Modulus → p];
   p2 = PolynomialRemainder[(p1 - x) * p2, fl, x, Modulus → p];
   (* p3 = fk-1(x) fk+1(x) *)
   p3 = f[k - 1] * f[k + 1];
   If[EvenQ[k],
    (* p16 = (x^(q^2) -x) fk^2(x) y^2 +fk-1(x) fk+1(x) k even *)
    p16x = p2 * y2 + p3;
    ,
    (* p16 = (x^(q^2) -x) fk^2(x) +fk-1(x) fk+1(x) y^2 k odd *)
    p16x = p2 + p3 * y2;
   ];
   p16x = PolynomialRemainder[p16x, fl, x, Modulus → 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 $w^2 \equiv q \,(\text{mod}\, l)$. If q has no square root mod l then we must have $t = 0$.

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

$p_{17}(x) = (x^q - x) f_w^2(x) \left(x^3 + a x + b\right) + f_{w-1}(x) f_{w+1}(x)$ w even

$p_{17}(x) = (x^q - x) f_w^2(x) + f_{w-1}(x) f_{w+1}(x) \left(x^3 + a x + b\right)$ w odd

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

```
ComputeEquation17[{a_, b_, p_}, l_, w_] :=
  Module[{ec1, y2, fl, p1, p2, p3, p17},
   ec1 = {a, b, p};
   y2 = x^3 + a x + b;
   fl = f[l];
   (* p2 = (x^q -x) f_w^2 (x) *)
   p1 = PolynomialPowerMod[x, p, {fl, p}];
   p2 = PolynomialMod[(p1 - x) * PolynomialMod[f[w]^2, {fl, p}], {fl, p}];
   (* p3 = f_w-1 (x) f_w+1 (x) *)
   p3 = f[w - 1] * f[w + 1];
   If[EvenQ[w],
    (* p(x) = (x^q -x) f_w^2 (x) y^2 + f_w-1 (x) f_w+1 (x) *)
    p17 = p2 * y2 + p3;
    ,
    (* p(x) = (x^q -x) f_w^2 (x) + f_w-1 (x) f_w+1 (x) y^2 *)
    p17 = p2 + p3 * y2;
   ];
   p17 = PolynomialMod[p17, {fl, p}];
   Return[p17];
  ];
```

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

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:

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

For w odd:

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

```
ComputeEquation18[{a_, b_, p_}, l_, w_] :=
  Module[{y2, fl, k, p1, p2, p3, p4, p18y},
   y2 = x^3 + a x + b;
   fl = 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 = (x^3 + a x + b)^k = (y^2)^k *)
   p1 = PolynomialPowerMod[y2, k, {fl, p}];
   (* p2 = f_w^3 (x) *)
   p2 = PolynomialRemainder[f[w]^2, fl, x, Modulus → p];
   p2 = PolynomialRemainder[f[w] * p2, fl, x, Modulus → p];
   (* p3 = f_{w+2} (x) f_{w-1}^2 (x) *)
   p3 = PolynomialRemainder[f[w - 1]^2, fl, x, Modulus → p];
   p3  = PolynomialRemainder[ f[w + 2] * p3, fl, x, Modulus → p];
   (* p4 = f_{w-2} (x)  f_{w+1}^2 (x) *)
   p4 = PolynomialRemainder[f[w + 1]^2, fl, x, Modulus → p];
   p4 = PolynomialRemainder[f[w - 2] * p4, fl, x, Modulus → p];
   (* p18y = 4y^{2k} f_w^3 (x) -f_{w+2} (x) f_{w-1}^2 (x) +f_{w-2} (x) f_{w+1}^2 (x) *)
   p18y = PolynomialRemainder[4 * p1 * p2 - p3 + p4, fl, x, Modulus → p];
   Return[p18y];
  ];
```

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

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

Compute α, 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

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

however we return α / y in this case and compensate at a higher level

for k odd

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

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

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

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

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

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

for k even

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

for k odd

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

however we return β / y in this case and compensate at a higher level

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

ComputeEquation19X[{*a*, *b*, *p*}, *l*]

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

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

Since we are not given τ, return the two terms

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

```
ComputeEquation19X[{a_, b_, p_}, l_] :=
  Module[{y2, fl, k, α2, β2, p1, p2, p3, p4, p5, p6, p7, p8, p19},
   k = Mod[p, l];
   fl = f[l];
   (* y^2 is a polynomial in x *)
   y2 = x^3 + a x + b;
   (* λ = α/β, note these are actually α/y for k even, β/y for k odd *)
   (* Note α,β are globals so we can share them with Equation19Y *)
   (* p1 = f_{k-1} f_{k+1} *)
   p1 = PolynomialRemainder[f[k - 1] * f[k + 1], fl, x, Modulus → p];
   (* p2 = x^p *)
   p2 = PolynomialPowerMod[x, p, {fl, p}];
   (* p3 = x^(p^2) *)
   p3 = PolynomialPowerMod[p2, p, {fl, p}];
   (* p4 = (x^(p^2) + x^p + x) *)
   p4 = p3 + p2 + x;
   (* p5 = f_k^2 *)
   p5 = PolynomialRemainder[f[k]^2, fl, x, Modulus → p];
   (* Compute α^2, β^2 compensating for single y's *)
   If[EvenQ[k],
    (* for k even α^2 = y^2 (α/y)^2 *)
    α2 = y2 * PolynomialRemainder[α^2, fl, x, Modulus → p];
    β2 = PolynomialRemainder[β^2, fl, x, Modulus → p];
    (* p7 = (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 *)
    p7 = PolynomialRemainder[p1 - y2 * p5 * p4, fl, x, Modulus → p];
    p7 = PolynomialRemainder[p7 * β2 + y2 * p5 * α2, fl, x, Modulus → p];
    ,
    (* for k odd β^2 = y^2 (β/y)^2 *)
    α2 = PolynomialRemainder[α^2, fl, x, Modulus → p];
    β2 = y2 * PolynomialRemainder[β^2, fl, x, Modulus → p];
    (* p7 = (y^2 f_{k-1} f_{k+1} - f_k^2 (x^(p^2)+x^p+x)) β^2 + f_k^2 α^2 *)
    p7 = PolynomialRemainder[y2 * p1 - p5 * p4, fl, x, Modulus → p];
    p7 = PolynomialRemainder[p7 * β2 + p5 * α2, fl, x, Modulus → p];
   ];
   p8 = PolynomialRemainder[β2 * p5, fl, x, Modulus → p];
   p19 = {p7, p8};
   Return[p19];
  ];
```

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

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

We are given the pre-computed terms, which do not depend on τ

$$p_7 = \beta^2\left(\psi_{k-1}\,\psi_{k+1} - \psi_k^2\left(x^{p^2} + x^p + x\right) + \alpha\,\psi_k^2\right),\ \ p_5 = \psi_k^2\,\beta^2$$

```
ComputeEquation19X[{a_, b_, p_}, l_, τ_, {p7_, p5_}] :=
 Module[{k, y2, fl, p6, p8, p9, p19},
  k = Mod[p, l];
  fl = f[l];
  (* y^2 is a polynomial in x *)
  y2 = x^3 + a x + b;
  (* p6 = (f_{τ-1} f_{τ+1})^p *)
  p6 = PolynomialPowerMod[f[τ - 1] * f[τ + 1], p, {fl, p}];
  p8 = PolynomialPowerMod[f[τ], 2 * p, {fl, p}];
  If[EvenQ[k] && EvenQ[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) y^{2p} f_τ^{2p} + f_{τ-1}^p f_{τ+1}^p β^2 y^2 f_k^2 *)
   p9 = PolynomialPowerMod[y2, p, {fl, p}];
   p19 = PolynomialRemainder[p7 * p9 * p8 + p6 * y2 * p5, fl, x, Modulus → 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 *)
   p9 = PolynomialPowerMod[y2, p + 1, {fl, p}];
   p19 = PolynomialRemainder[p7 * p8 + p9 * p6 * p5, fl, x, Modulus → 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 *)
   p9 = PolynomialPowerMod[y2, p, {fl, p}];
   p19 = PolynomialRemainder[p7 * p9 * p8 + p6 * p5, fl, x, Modulus → 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 *)
   p9 = PolynomialPowerMod[y2, p, {fl, p}];
   p19 = PolynomialRemainder[p7 * p8 + p9 * p6 * p5, fl, x, Modulus → p];
  ];
  Return[p19];
 ]
```

ComputeEquation19X[{a, b, p}, l, τ]

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

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

```
ComputeEquation19X[{a_, b_, p_}, l_, τ_] :=
  Module[{y2, fl, k, α2, β2, p1, p2, p3, p4, p5, p6, p7, p8, p9, p19},
   k = Mod[p, l];
   fl = f[l];
   (* y^2 is a polynomial in x *)
   y2 = x^3 + a x + b;
   (* λ = α/β, note these are actually α/y for k even, β/y for k odd *)
   (* Note α,β are globals so we can share them with Equation19Y *)
   (* p1 = f_{k-1}f_{k+1} *)
   p1 = f[k - 1] * f[k + 1];
   (* p2 = x^p *)
   p2 = PolynomialPowerMod[x, p, {fl, p}];
   (* p3 = x^(p^2) *)
   p3 = PolynomialPowerMod[p2, p, {fl, p}];
   (* p4 = (x^(p^2) + x^p + x) *)
   p4 = p3 + p2 + x;
   (* p5 = f_k^2 *)
   p5 = PolynomialPowerMod[f[k], 2, {fl, p}];
   (* p6 = (f_{τ-1}f_{τ+1})^p *)
   p6 = PolynomialPowerMod[f[τ - 1] * f[τ + 1], p, {fl, p}];
   (* Compute α^2, β^2 compensating for single y's *)
   If[EvenQ[k],
    (* for k even α^2 = y^2(α/y)^2 *)
    α2 = y2 * PolynomialPowerMod[α, 2, {fl, p}];
    β2 = PolynomialPowerMod[β, 2, {fl, p}];
    (* p7 = (f_{k-1}f_{k+1} - y^2f_k^2(x^(p^2)+x^p+x))β^2+y^2f_k^2α^2 *)
    p7 = PolynomialPowerMod[p1 - y2 * p5 * p4, 1, {fl, p}];
    p7 = PolynomialPowerMod[p7 * β2 + y2 * p5 * α2, 1, {fl, p}];
    ,
    (* for k odd β^2 = y^2(β/y)^2 *)
    α2 = PolynomialPowerMod[α, 2, {fl, p}];
    β2 = y2 * PolynomialPowerMod[β, 2, {fl, p}];
    (* p7 = (y^2f_{k-1}f_{k+1} - f_k^2(x^(p^2)+x^p+x))β^2+f_k^2α^2 *)
    p7 = PolynomialPowerMod[y2 * p1 - p5 * p4, 1, {fl, p}];
    p7 = PolynomialPowerMod[p7 * β2 + p5 * α2, 1, {fl, p}];
   ];
   If[EvenQ[k] && EvenQ[τ],
    (* ((f_{k-1}f_{k+1} - y^2f_k^2(x^(p^2)+x^p+x))β^2 + y^2f_k^2α^2)y^(2p)f_τ^(2p) + f_{τ-1}^p f_{τ+1}^p β^2y^2f_k^2 *)
    p8 = PolynomialPowerMod[f[τ], 2 * p, {fl, p}];
```

```
  p9 = PolynomialPowerMod[y2, p, {fl, p}];
  p19 = PolynomialPowerMod[p7 * p9 * p8 + p6 * β2 * y2 * p5, 1, {fl, 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, {fl, p}];
  p9 = PolynomialPowerMod[y2, p + 1, {fl, p}];
  p19 = PolynomialPowerMod[p7 * p8 + p9 * p6 * β2 * p5, 1, {fl, 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, {fl, p}];
  p9 = PolynomialPowerMod[y2, p, {fl, p}];
  p19 = PolynomialPowerMod[p7 * p9 * p8 + p6 * β2 * p5, 1, {fl, 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, {fl, p}];
  p9 = PolynomialPowerMod[y2, p, {fl, p}];
  p19 = PolynomialPowerMod[p7 * p8 + p9 * p6 * β2 * p5, 1, {fl, p}];
 ];
 Return[p19];
];
```

ComputeEquation19Y[{*a*, *b*, *p*}, *l*, τ]

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

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

```
ComputeEquation19Y[{a_, b_, p_}, l_, τ_] :=
 Module[{y2, fl, k, a2, a3, b2, b3, y4, yq, p1, p2, p3, p4, p5, p6, p7, p19},
  fl = f[l];
  k = Mod[p, l];
  (* y^2 is a polynomial in x only *)
  y2 = x^3 + a x + b;
  (* λ = α/β, note these are actually α/y for k even, β/y for k odd *)
  (* Compute α^3, β^2, β^3 *)
  a3 = PolynomialPowerMod[α, 3, {fl, p}];
  b2 = PolynomialMod[β^2, {fl, p}];
  b3 = PolynomialMod[β * b2, {fl, p}];
  (* p1 = α β^2 (f_{k-1} f_{k+1}) *)
  p1 = PolynomialMod[α * b2 * f[k - 1] * f[k + 1], {fl, p}];
  (* p2 = f_k^2 *)
  p2 = PolynomialMod[f[k]^2, {fl, p}];
  (* p3 = 2 x^(p^2) + x *)
  p3 = PolynomialPowerMod[x, p, {fl, p}];
  p3 = PolynomialPowerMod[p3, p, {fl, p}];
  p3 = PolynomialMod[2 * p3 + x, {fl, p}];
  (* p4 = 4 f_τ^(3p) = (4 f_τ^3)^p *)
  p4 = 4 * PolynomialPowerMod[f[τ], 3 p, {fl, p}];
  (* p5 = (f_{τ-1}^2 f_{τ+2} - f_{τ-2} f_{τ+1}^2)^p *)
  p5 = f[τ + 2] * PolynomialMod[f[τ - 1]^2, {fl, p}];
  p5 = p5 - f[τ - 2] * PolynomialMod[f[τ + 1]^2, {fl, p}];
  p5 = PolynomialMod[p5, {fl, p}];
  p5 = PolynomialPowerMod[p5, p, {fl, p}];
  (* p6 = y^(p^2-1) = (y^2)^((p-1)(p+1)/2) *)
  p6 = PolynomialPowerMod[y2, (p - 1), {fl, p}];
  p6 = PolynomialPowerMod[p6, (p + 1) / 2, {fl, p}];
  Print["{k,τ} = ", {k, τ}];
  If[EvenQ[k],
   (* 4^q f_τ^(3q) (((2 x^(p^2)+x) α β^2 - β^3 y^(p^2-1) - y^2 α^3) y^2 f_k^2 - α β^2 f_{k-1} f_{k+1} ) *)
   p19 = PolynomialMod[p3 * α * b2 - b3 * p6 - y2 * a3, {fl, p}];
   p19 = PolynomialMod[p4 * (p19 * y2 * p2 - p1), {fl, p}];
   If[EvenQ[τ],
    (* 4^p f_τ^(3p) y^(3p+1) (((2 x^(p^2)+x) α β^2 - β^3 y^(p^2-1) - y^2 α^3) y^2 f_k^2 -α β^2 f_{k-1} f_{k+1} ) -
     β^3 y^2 f_k^2 (f_{τ-1}^2 f_{τ+2}-f_{τ-2} f_{τ+1}^2)^p *)
```

```
    p19 *= PolynomialPowerMod[y2, (3 p + 1) / 2, {fl, p}];
    p19 = PolynomialMod[p19 - b3 * y2 * p2 * p5, {fl, p}];
    ,
    (* 4^p f_τ^(3p) (((2 x^(p^2)+x) α β^2 - β^3 y^(p^2-1) - y^2 α^3) y^2 f_k^2 - α β^2 f_(k-1) f_(k+1)) -
     β^3 f_k^2 (f_(τ-1)^2 f_(τ+2) - f_(τ-2) f_(τ+1)^2)^p y^(p+1)
    *)
    yq = PolynomialPowerMod[y2, (p + 1) / 2, {fl, p}];
    p19 = PolynomialMod[p19 - b3 * p2 * p5 * yq, {fl, p}];
   ];
  ];
  If[OddQ[k],
   y4 = PolynomialPowerMod[y2, 2, {fl, p}];
   (* p_7 = β^3 (y^2)^((p^2+3)/2) = β^3 (y^2)^2 (y^2)^((p^2-1)/2) *)
   p7 = PolynomialMod[b3 * y4 * p6, {fl, p}];
   p19 = PolynomialMod[p3 * α * b2 * y2 - p7 - a3, {fl, p}];
   p19 = PolynomialMod[p4 * (p19 * p2 - y4 * p1), {fl, p}];
   If[EvenQ[τ],
    (* 4^p f_τ^(3p) y^(3p-3) (((2 x^(p^2)+x) α β^2 y^2 - β^3 y^(p^2+3) - α^3) f_k^2 - y^4 α β^2 f_(k-1) f_(k+1)) -
     β^3 f_k^2 (f_(τ-1)^2 f_(τ+2) - f_(τ-2) f_(τ+1)^2)^p *)
    p19 *= PolynomialPowerMod[y2, (3 p - 3) / 2, {fl, p}];
    p19 = PolynomialMod[p19 - b3 * p2 * p5, {fl, p}];
    ,
    (* 4 f_τ^(3p) (((2x^(p^2)+x) α β^2 y^2 - β^3 y^(p^2+3) - α^3) f_k^2 - α β^2 y^4 f_(k-1) f_(k+1)) -
     β^3 f_k^2 (f_(τ-1)^2 f_(τ+2) - f_(τ-2) f_(τ+1)^2)^p y^(p+3) *)
    (* 4^p f_τ^(3p) (((x^(p^2)+x) α β^2 y^2 - β^3 y^(p^2+3) - α^3) f_k^2 - α y^4 β^2 f_(k-1) f_(k+1)) -
     β^3 f_k^2 (f_(τ-1)^2 f_(τ+2) - f_(τ-2) f_(τ+1)^2)^p y^(p+3) *)
    yq = PolynomialPowerMod[y2, (p + 3) / 2, {fl, p}];
    p19 = PolynomialMod[p19 - b3 * p2 * p5 * yq, {fl, p}];
   ];
  ];
  Return[p19];
 ];
```

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

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[{a_, b_, p_}, l_] := Module[{ec1, w, fl, g, p17, p18, tl},
  ec1 = {a, b, p};
  (* 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]; *)
  fl = f[l];
  (* Print["f[",l,"] = ",fl]; *)
  (* Use Schoof 17 to test ϕ1P = ±w*P (x coord) *)
  p17 = ComputeEquation17[ec1, l, w];
  (* Print["p17 = ",p17]; *)
  g = PolynomialGCD[p17, fl, 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 ϕ1P = w*P (y coord) *)
  p18 = ComputeEquation18[ec1, l, w];
  (* Print["p18 = ",p18]; *)
  g = PolynomialGCD[p18, fl, Modulus → p];
  (* Print["gcd(p18,f1) = ",g]; *)
  (* g = 1 means no such P, so t = -2w, else t = 2w *)
  If[SameQ[g, 1], tl = Mod[-2 w, l], tl = Mod[2 w, l]];
  Return[tl];
 ];
```

EcGroupOrderSchoof[{*a*, *b*, *p*}]

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

1. If $\gcd(x^3 + a x + b,\ x^p - x) = 1$ 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 ψ_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 $t \equiv 2 w \pmod l$, else $t \equiv -2 w \pmod l$.
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 τ
16. Next l
17. At this point we have computed $t \pmod{l_i}$ for all $l_i \in S$,
18. now 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.

```
EcGroupOrderSchoof[{a_, b_, p_}] :=
  Module[{e, ec1, t0, prmprod, pr2, h0, c, g, t, l, s, m, i, p5, p7,
     p16, p17, p18, p19x, τ, p19y, gy, r},
    ec1 = {a, b, p};
    e = x^3 + a x + b;
    Print["E(", Subscript["F", p], "):", TraditionalForm[e]];
    t0 = AbsoluteTime[];
    pr2 = Floor[2 * Sqrt[p]];
    h0 = p + 1 - pr2; h1 = p + 1 + pr2;
    Print["Hasse's bounds ", h0, " ≤ #E(Fp) ≤ ", h1];
    (* Determine t mod 2 *)
    c = PolynomialPowerMod[x, p, {e, p}] - x;
    (* Print["x^p-x ≡ ",c," mod(E,p)"]; *)
```

```
    g = PolynomialGCD[c, e, Modulus → p];
    (* Print["gcd(x^p-x,E) = ",g]; *)
    t = {{2, If[SameQ[g, 1], 1, 0]}};
    {prmprod, s} = ComputePrimeSet[p];
    Print[s];
    m = Last[s] + 2;
    Print["Computing ", m, " division polynomials"];
    ψ = ComputeDivisionPolynomials[ec1, m];
    For[i = 2, i ≤ Length[s], ++i,
     l = s[[i]];
     fl = f[l];
     (* Print ["f[",l,"] = ",fl]; *)
     (* If ∃ P ∈ E[l] ∋ ϕ_q^2P = ±kP we are in case 1 *)
     p16 = ComputeEquation16[ec1, l];
     (* Print["p16 = ",p16]; *)
     g = PolynomialGCD[p16, fl, Modulus → p];
     (* Print["gcd(p16,f1) = ", g]; *)
If[! SameQ[g, 1],
  (* Print["Case 1"]; *)
  (* Determine t (mod l) using Schoof (17) and (18) *)
  tl = ComputeTmodLCaseOne[ec1, l];
  AppendTo[t, {l, tl}];
  Print[AbsoluteTime[] - t0];
  ,
  (* Else, we are in case 2 testing ϕ_q^2P + kP = ±τϕ_qP *)
  (* Print["Case 2"]; *)
  (* Determine if ∃ τ < l satisfying Schoof (19) *)
  α = ComputeAlpha[ec1, l];
  β = ComputeBeta[ec1, l];
  {p7, p5} = ComputeEquation19X[ec1, l];
  (* Print["α = ",α]; *)
  (* Print["β = ",β]; *)
  For[τ = 1, τ ≤ (l + 1) / 2, ++τ,
   (* Print["τ1 = ",τ]; *)
   (* If (19X) holds for some x ∈ E[l] *)
   p19x = ComputeEquation19X[ec1, l, τ, {p7, p5}];
   (* Print["p19x = ",p19x]; *)
   g = PolynomialGCD[p19x, fl, Modulus → p];
   (* Print["gcd(p19x,f1) = ", g]; *)
   If[! SameQ[g, 1],
    (* then ϕ_q^2P + kP = ±τϕ_qP, use (19y) to determine the sign *)
    p19y = ComputeEquation19Y[ec1, l, τ];
    (* Print["p19y = ",p19y]; *)
    gy = PolynomialGCD[p19y, fl, Modulus → p];
    (* Print["gcd(p19y,f1) = ", gy]; *)
    If[! SameQ[gy, 1], tl = τ, tl = l - τ];
```

```
                AppendTo[t, {l, tl}];
                Print[AbsoluteTime[] - t0];
                (* There can be only one such τ for each l *)
                Break[];
              ];
            ];
          ];
        Print["τ (mod ", l, " ) = ", t];
        ];
      (* Convert t from {{l, t (mod l)} to {{l},{tl}} *)
      t = Transpose[t];
      (* Use the Chinese Remainder Theorem to compute the smallest
       r > 0 such that r ≡ t_i (mod l_i) for each i *)
      r = ChineseRemainder[Last[t], First[t]];
      (* force -2√p ≤ r ≤ 2√p *)
      If[r > 2 √p , r -= prmprod];
      Print["τ = ", r];
      grpord = p + 1 - r;
      Return[grpord];
      ];
```

EcGroupOrderSchoofCont[{a, b, p}, tp]

Compute the order of the group of points on the elliptic curve

$E: y^2 \equiv x^2 + a\,x + b \,(\mathrm{mod}\, p)$

Given that $\sharp\, E(\mathbb{F}_p) = p + 1 - \tau$

and we already know that $\tau \equiv t_i \bmod l_i$ for $i = 1$ to m

and tp = $\{\{l_1, t_1\}, \{l_2, t_2\}, \ldots, \{l_m, t_m\}\}$

```
EcGroupOrderSchoofCont[{a_, b_, p_}, tp_] :=
  Module[{e, ec1, t0, prmprod, pr2, h0, c, g, t, l, s, m, i, p5, p7, p16,
    p17, p18, p19x, τ, p19y, gy, r},
   If[Length[tp] == 0, Return[EcGroupOrderSchoof[{a, b, p}]]];
   t = tp;
   ec1 = {a, b, p};
   e = x^3 + a x + b;
   Print["E(", Subscript["𝔽", p], "):", TraditionalForm[e]];
   t0 = AbsoluteTime[];
   pr2 = Floor[2 * Sqrt[p]];
   h0 = p + 1 - pr2; h1 = p + 1 + pr2;
   Print["Hasse's bounds ", h0, " ≤ #E(𝔽p) ≤ ", h1];
   {prmprod, s} = ComputePrimeSet[p];
   Print[s];
   m = Last[s] + 2;
   Print["Computing ", m, " division polynomials"];
   ψ = ComputeDivisionPolynomials[ec1, m];
```

```
For[i = Length[tp] + 1, i ≤ Length[s], ++i,
 l = s[[i]];
 fl = f[l];
 (* Print ["f[",l,"] = ",fl]; *)
 (* If ∃ P ∈ E[l] ∋ ϕ_q^2P = ±kP we are in case 1 *)
 p16 = ComputeEquation16[ec1, l];
 (* Print["p16 = ",p16]; *)
 g = PolynomialGCD[p16, fl, Modulus → p];
 (* Print["gcd(p16,f1) = ",g]; *)
 If[! SameQ[g, 1],
  (* Print["Case 1"]; *)
  (* Determine t (mod l) using Schoof (17) and (18) *)
  tl = ComputeTmodLCaseOne[ec1, l];
  AppendTo[t, {l, tl}];
  Print[AbsoluteTime[] - t0];
  ,
  (* Else, we are in case 2 testing ϕ_q^2P + kP = ±τϕ_qP *)
  (* Print["Case 2"]; *)
  (* Determine if ∃ τ < l satisfying Schoof (19) *)
  α = ComputeAlpha[ec1, l];
  β = ComputeBeta[ec1, l];
  {p7, p5} = ComputeEquation19X[ec1, l];
  (* Print["α = ",α]; *)
  (* Print["β = ",β]; *)
  For[τ = 1, τ ≤ (l + 1) / 2, ++τ,
   (* Print["τ1 = ",τ]; *)
   (* If (19X) holds for some x ∈ E[l] *)
   p19x = ComputeEquation19X[ec1, l, τ, {p7, p5}];
   (* Print["p19x = ",p19x]; *)
   g = PolynomialGCD[p19x, fl, Modulus → p];
   (* Print["gcd(p19x,f1) = ", g]; *)
   If[! SameQ[g, 1],
    (* then ϕ_q^2P + kP = ±τϕ_qP, use (19y) to determine the sign *)
    p19y = ComputeEquation19Y[ec1, l, τ];
    (* Print["p19y = ",p19y]; *)
    gy = PolynomialGCD[p19y, fl, Modulus → p];
    (* Print["gcd(p19y,f1) = ", gy]; *)
    If[! SameQ[gy, 1], tl = τ, tl = l - τ];
    AppendTo[t, {l, tl}];
    Print[AbsoluteTime[] - t0];
    (* There can be only one such τ for each l *)
    Break[];
   ];
  ];
 ];
 Print["τ (mod ", l, " ) = ", t];
```

```
  ];
  (* Convert t from {{l, t (mod l)} to {{l},{tl}} *)
  t = Transpose[t];
  (* Use the Chinese Remainder Theorem to compute the smallest
   r > 0 such that r ≡ t_i (mod l_i) for each i *)
  r = ChineseRemainder[Last[t], First[t]];
  (* force -2√p ≤ r ≤ 2√p *)
  If[r > 2 √p , r -= prmprod];
  Print["τ = ", r];
  grpord = p + 1 - r;
  Return[grpord];
];
```

Test Sections