Final Report

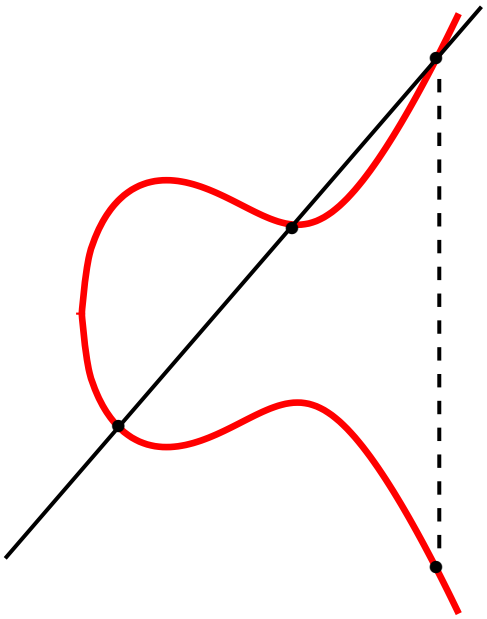
Cryptography and Number Theory Boot Camp NSF-REU

Angel Agüero [*] Mahmoud El-Kishky [†] Dietrich Jenkins [‡]
Catherine Marin King[†] Asa Linson[†] Enrique Salcido[*]
Kaitlin Tademy [§]

Summer 2017

Abstract

In this manuscript, we describe several cryptosystems in detail, and detail the mathematics behind them. This includes the extensive application of topics from abstract algebra and number theory, including a detailed study of elliptic curves. Finally, we introduce a new cryptosystem based on the theory of elliptic curves, carefully describing what makes it mathematically secure.



[*]University of Texas at El Paso
[†]University of Texas at Tyler
[‡]University of Kansas
[§]Sam Houston State University

Contents

Introduction

Modular arithmetic

Modular arithmetic, the theory of congruence modulo an integer, is vital to the functionality of many cryptosystems. Given integers a and b, and a positive integer n, we say that a *is congruent to* b *modulo* n, and write

$$a \equiv b \bmod n,$$

if $n \mid (b-a)$. Notice that $a \equiv 0 \bmod n$ if and only if $n \mid a$.

Modular arithmetic relies on four basic properties, which make it an *equivalence relation* on $\mathbb{Z}$.

Proposition 1. *For all integers a, b, and c, and a positive integer n, the following hold.*

(1) (Reflexivity) $a \equiv a \bmod n$.

(2) (Symmetry) $a \equiv b \bmod n$ *if and only if* $b \equiv a \bmod n$.

(3) (Transitivity) *If* $a \equiv b \bmod n$ *and* $b \equiv c \bmod n$, *then* $a \equiv c \bmod n$.

In addition to these properties, the following facts help us perform arithmetic modulo n.

Proposition 2. *Given integers a, b, c, and d, and a positive integer n, if $a \equiv b \bmod n$ and $c \equiv d \bmod n$, then*

(1) $a \pm c \equiv b \pm d \bmod n$, *and*

(2) $ac \equiv bd \bmod n$.

Division in modular arithmetic is more complex. Conceptually, division for real numbers can be thought of as multiplying by a reciporical; this motivates the analogue for modular arithmetic. Given integers a and $n > 0$, an integer b is the *multiplicative inverse* of a modulo n if $ab \equiv 1 \bmod n$. An integer a has a multiplicative inverse modulo n if and only if a and n are relatively prime. Multiplicative inverses are unique modulo n in the sense that any two multiplicative inverses are congruent modulo n.

To find a multiplicative inverse of a modulo n, we know that $(a, n) = 1$, and we can find such an inverse using the extended Euclidean Algorithm. We illustrate this method via the following example.

Example 3. Given the modulus $n = 44$ and the integer $a = 35$, the Euclidean Algorithm is performed as follows:

$$\begin{aligned} 44 &= 35 \cdot 1 + 9 \\ 35 &= 9 \cdot 3 + 8 \\ 9 &= 8 \cdot 1 + 1 \end{aligned}$$

Since the last nonzero remainder equals one, this verifies that a and n are relatively prime.

From here, using back substitution, we can find integers x and y for which $ax+ny = 1$, which are ensured to exist by Bézout's Theorem.

$$\begin{aligned} 1 &= 9 - 8 \cdot 1 \\ 1 &= 9 - (35 - 9 \cdot 3) \cdot 1 \\ 1 &= 9 \cdot 4 - 35 \cdot 1 \\ 1 &= (44 - 35 \cdot 1) \cdot 4 - 35 \cdot 1 \\ 1 &= 44 \cdot 4 - 35 \cdot 5 \end{aligned}$$

We conclude that $x = 4$ and $y = -5$. Moreover, this equation tells us that

$$35 \cdot -5 \equiv 1 \bmod 44,$$

and we see that $-5 \equiv 39 \bmod 44$ is the multiplicative inverse of 35 modulo 44.

Remark 4. The greatest common divisor of two integers is the largest divisor of both numbers. The Euclidean algorithm is one of most effective means of finding a greatest common divisor. For this reason, this algorithm is particularly relevant to our study of cryptography. For example, in cryptosystems such as RSA, multiplicative inverses are used critically in both securing communications via insecure channels, and in breaking these same types of codes.

Public key cryptography

Public key cryptography refers to the process of securely sending messages with an encryption key that is made public. In order for this type of message exchange to be successful, some information is public, while other information must be kept private. It is necessary to strategically choose which information is made public, and which is kept private so that it is nearly impossible to break the cryptosystem using only public information. Anyone may encrypt a message using the public key(s), but for a message to be decoded, one must use a private decryption key known only to the receiver.

The goal is for encryption to be rather simple but decryption to be computationally difficult: the security in such a cryptosystem typically relies on a mathematical process that is relatively easy to perform, but for which there is no known way to invert efficiently. In ideal cases, the time taken to break a system should be astronomical.

In describing a public key cryptosystem, we often represent the the sender by the name "Alice," and the receiver by "Bob," respectively, and use the name "Eve" to represent an entity that may eavesdrop on the message in the public channel through which data is passed. We refer the reader to the texts [2] and [3] as references for the cryptosystems we describe.

The RSA cryptosystem

The RSA Cryposystem is named after its creators: Rivest, Shamir, and Adleman. It is an example of a public key cryptosystem, so that the data needed to encrypt a

message is public, so anyone can encrypt a message. On the other hand, only one party, Alice, has the private information necessary to decrypt messages.

To begin, Alice chooses two large, distinct primes p and q with product $n = pq$. If φ denotes Euler's phi function, which computes the number of positive integers less than a given integer that are relatively prime to it (therefore, counting the units modulo that integer), Alice can easily calculate

$$\varphi(n) = \varphi(pq) = \varphi(p)\varphi(q) = (p-1)(q-1).$$

She selects any unit modulo $\varphi(n)$, and calls it e; this will be the "encryption exponent." She then uses the Euclidean Algorithm to find the inverse of e modulo n, and calls it d; this will be the "decryption exponent."

Alice then transmits the pubic data, which consists of the pair (n, e), via the unsecured public channel. Encryption is given by following instructions: To encrypt a message, break it into blocks of size less than n. To encrypt a block m, transmit $0 \leq y < n$, where $y \equiv m^e \bmod n$.

To decrypt a message, Alice raises the encrypted message to the decryption exponent d modulo n:

$$y^d \equiv (m^e)^d \equiv m^{ed} \bmod n.$$

Since $ed \equiv 1 \bmod \varphi(n)$, we know that $ed = 1 + \varphi(n)k$ for some $k \in \mathbb{Z}$. By substitution, we see that

$$m^{ed} \equiv m^{(1+\varphi(n)k)} \equiv m \cdot m^{\varphi(n)k} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \cdot 1^k \equiv m \bmod n,$$

where the second to last congruence holds by Euler's Theorem.

The security of RSA lies in the difficulty of factoring the large integer n, i.e., finding the primes p and q, given n. For example, if p and q are both approximately 150 digits, then with current computational power, it would take approximately 600,000 years to factor n!

Discrete logarithm problem

Like factoring large numbers with large factors, the *discrete logarithm problem* is computationally very difficult, and is used in other public key cryptosystems: solving for x in the equation

$$\beta \equiv \alpha^x \bmod p,$$

where α and β are nonzero integers, and p is a prime. Usually, α is chosen to be a primitive root modulo p, so that a solution x must exist. However, if p is very large, it is typically very difficult to find x. The discrete logarithm problem is applied in the *Diffie-Hellman public key exchange*, and the related *ElGamal encryption scheme*, which we now describe.

Diffie-Hellman key exchange

The goal of the Diffie-Hellman key Exchange is for the two parties, Alice and Bob, to agree on a secret key by only passing data through a public channel. Implementation

of the Diffie-Hellman key Exchange begins with first choosing a large prime number p, which will serve as modulus, and g, a primitive root modulo p. The integers p and g are made public.

Next, Alice chooses a secret positive integer x. Alice then calculates the least non-negative residue of χ modulo p:

$$\chi \equiv g^x \bmod p.$$

The integer χ is then sent over the public channel to Bob, who keeps it for future calculations. Meanwhile, Bob chooses a positive integer y, which he keeps secret, and calculates the least non-negative residue

$$\mathcal{Y} \equiv g^y \bmod p.$$

The he sends $\mathcal{Y}$ across the public channel to Alice.

Alice now computes the key k as the least non-negative residue of $\mathcal{Y}^x$ modulo p. Notice that

$$k \equiv \mathcal{Y}^x \equiv (g^y)^x \equiv g^{xy} \bmod p,$$

which means that Bob to can also calculate k as

$$\chi^y \equiv (g^x)^y \equiv g^{xy} \equiv k \bmod p.$$

Therefore, Alice and Bob have agreed on a secret key k, whose privacy relies on the difficulty of the discrete logarithm problem.

ElGamal encryption

The Diffie-Helleman key exchange is the basis for the ElGamal encryption scheme. The distinction between Diffie-Hellman and ElGamal comes from ElGamal's ability to transfer a message based on the key shared by the sender and receiver via Diffie-Hellman.

Suppose that the parties agree on the key k using Diffie-Hellman. To describe encryption using ElGamal, suppose we aim to transmit the plaintext message m. To encrypt, we multiply m by the key k, and find its least non-negative residue modulo the fixed public prime number $p > k$. This gives you the encrypted message n, where

$$n \equiv mk \bmod p.$$

To decrypt a message, first find the multiplicative inverse of the key k modulo p, which can be done using the Extended Euclidean Algorithm. Call the inverse ℓ, so that

$$\ell k \equiv 1 \bmod p.$$

Then to see that encrypting, and then decrypting results in the original message, we calculate that

$$\ell n \equiv \ell(mk) \equiv (\ell k)m \equiv m \bmod p.$$

Factoring

Many cryptosystems, such as RSA, rely on the fact that it is significantly computationally difficult to factor a large composite, especially when it is the product of large primes. Given a composite integer that we would like to factor non-trivially, there are several factoring methods that can be used. In order to build a secure cryptosystem that relies on the difficulty of factoring, it is be imperative to chose an integer that is difficult to factor, even using the information in the cryptosystem that is made public.

In this paper, we will discuss three factoring methods: the naive method, Pollard's $p-1$ factoring algorithm, and Lenstra's elliptic curve factoring algorithm. Note that we wait to discuss the last factoring method until Subsection 3.5, since it relies on the theory of groups defined by elliptic curves.

Naive factoring method

To factor an integer $n > 1$, the naive factoring method is implemented by checking if any positive integer from 1 to $\lfloor\sqrt{n}\rfloor$ divides n. This is a sure method that will either find a factor of n, or show that n is prime. For example, if we want to find a factor of 26, then we check whether each positive integer from 1 to 5 is a factor, using the division algorithm. In doing so, we will find that 2 divides 26.

Although the naive methods always results in a factor if n is composite, it becomes quite tedious and inefficient for n large; for example, if n an 100-digit integer. Therefore, it is pertinent that we use a more efficient method to factor large composite integers, if possible.

Pollard's $p-1$ factoring algorithm

The goal of Pollard's $p-1$ factoring algorithm is to find a non-trivial factor a positive composite integer n, and is performed as follows:

1. Choose a small integer $1 < a < n$.
2. Find $\gcd(a, n)$.
    - If $\gcd(a, n) \neq 1$, then $\gcd(a, n) \mid n$. In other words, $\gcd(a, n)$ is a nontrivial factor of n.
    - If $\gcd(a, n) = n$, a trivial factor, the algorithm terminates.
    - Otherwise, $\gcd(a, n) = 1$, again a trivial factor. In this case, recursively

calculate the following sequence:

$$\begin{aligned} a_1 &\equiv a \bmod n \\ a_2 &\equiv a_1^2 \bmod n \\ a_3 &\equiv a_2^3 \bmod n \\ a_4 &\equiv a_3^4 \bmod n \\ &\vdots \\ a_i &\equiv a_{i-1}^i \bmod n \end{aligned}$$

- Find $\gcd(a_i - 1, n)$ for each i until $\gcd(a_i - 1, n) \neq 1$ or n for some $i > 0$.

3. If p is a prime factor of n and B is a positive integer for which $(p-1) \mid B!$, then this algorithm terminates in at most B steps. However, it is possible that the greatest common divisor computed equals n, so that we do not obtain a non-trivial factor of n. In this case, we can choose a new a value and repeat the process.

For example, let $n = 901$.

1. We choose $a = 2$.
2. We calculate $\gcd(2, 901) = 1$.
3. Then, we find $a_2 \equiv 2^2 \equiv 4 \mod 901$, and calculate that $\gcd(3, 901) = 1$.
4. Next, we find $a_3 \equiv 4^3 \equiv 64 \mod 901$ and calculate that $\gcd(63, 901) = 1$.
5. Finally, we find $a_4 \equiv 64^4 \equiv 596 \mod 901$ and calculate that $\gcd(595, 901) = 17 \neq 1$. Thus, 17 is a factor of 901, and we can factor 901 as
$$901 = 17 \cdot 53.$$
6. Note that since $p = 17$, $p - 1 = 16$ and $(p-1) \mid B!$, where $B = 6$. This verifies that the algorithm terminated in $4 < 6 = B$ steps.

Due to the number of steps required, in general, Pollard's $p-1$ Algorithm is quicker and more efficient than the naive factoring method.

Elliptic curves

An elliptic curve is a curve defined by an equation of the form

$$y^2 = x^3 + ax^2 + bx + c, \tag{3.1}$$

where the coefficients a, b, and c are elements of some fixed ambient field. When working over the rational numbers $\mathbb{Q}$ or the real numbers $\mathbb{R}$, these curves can be

plotted on the $xy-$plane, and when considered in this way, the resulting curves are symmetric about the x-axis due to the y^2 term on the equation's left-hand side.

The main goal in this section is to use an elliptic curve to create an abelian group. The set of elements E of the group consists of all points (x, y) satisfying the equation (3.1), along with an additional element denoted by the symbol "∞," and often called the "point at infinity." Shortly, we will explicitly define an operation on E, which we call addition and likewise denote "$+$." This operation satisfies the group axioms, and makes the element ∞ the group's additive identity.

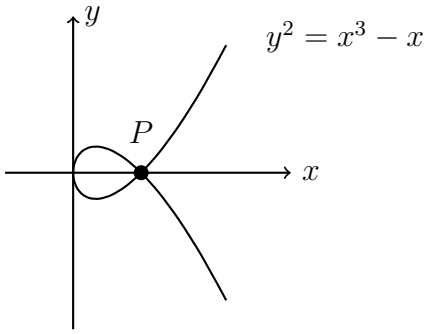
Roughly, adding two points in E that are solutions to equation (3.1) (that is, that are not ∞) amounts to finding the unique line determined by these two points, and then solving for the third point on the line that also intersects the curve. The sum of the two points in question is defined as the additive inverse of the third point on the intersection of the line and the curve.

In this discussion, it is important to notice that almost every pair of *distinct* points on the curve determine a line that passes through the curve at a third point, after one accounts for multiplicities. Indeed, the only exceptions to this are pairs of points that are reflections of one another other across the x-axis. However, it turns out that the sum of such points is defined in a particularly simple way.

We emphasize the term "distinct" in the previous paragraph, so that it makes sense to refer to the unique line through both points. When instead the two points are equal, a case we encounter in adding a point to itself, our alternative is to consider the tangent line to the curve at this point. However, this breaks down when the curve given by (3.1) has a *singular point*, a points for which there exists no unique tangent line. A curves that contain a singular point is called a *singular curve*. An elliptic curve is singular precisely if its *discriminant* $\Delta = a^2b^2 - 4a^3c - 27c^2 + 18abc$ (with coefficients as in (3.1)) is zero. Note that the discriminant is the square of the product of the differences of all pairs of roots of the right-hand side of the equation (3.1), so a curve is singular if and only if the right-hand side has a double (or triple) root.

Throughout our discussion of elliptic curves in this manuscript, we consider only non-singular elliptic curves, although we could simply remove the singular point from the set E, and the remaining elements form a group under the same addition conventions.

In the figure below, we illustrate a singular curve with singular point P. Notice that there is no unique tangent line to the graph at this point.



As noted above, the identity element of the group on the set E is defined to be the element $\infty \in E$. Geometrically, if we project the $xy-$plane onto a sphere, the point at which the sphere closes can be thought of as the "point at infinity," and we can think of this as a point lying on any vertical line. Since ∞ is the identity of the group, $P + \infty = P$ for every $P \in E$, and $\infty = -\infty$.

The additive inverse of a point $P = (x, y)$ on E satisfying equation (3.1) is obtained from P by negating the y coordinate of the point; i.e., $-P = (x, -y)$. Geometrically, this amounts to reflecting P over the x-axis. For example, consider the point $P = (1, 3)$ on the elliptic curve defined by the equation $y^2 = x^3 + 8$. Then $-P = (1, -3)$, which also satisfies this equation.

For this definition of additive inverses to be consistent with the group axioms, given any point $P \in E$, then we must have that $P + (-P) = \infty$. Geometrically, the line passing through two points that are additive inverses is vertical. Therefore, this line intersects the curve in only these two points, so that the other point it "intersects" must be the "point at infinity," our group's additive identity.

Addition law on an elliptic curve

It remains to precisely define the sum of two arbitrary points on an elliptic curve E, when neither point is the additive identity ∞. In what follows, we describe addition via three cases, providing equations to calculate the resulting point. In each case, we include a figure illustrating the addition geometrically.

Throughout, we make the convention that $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ are points on an elliptic curve E given by equation (3.1).

Case 1: $x_P \neq x_Q$.

In this setting, first find the unique line that intersects these two points,

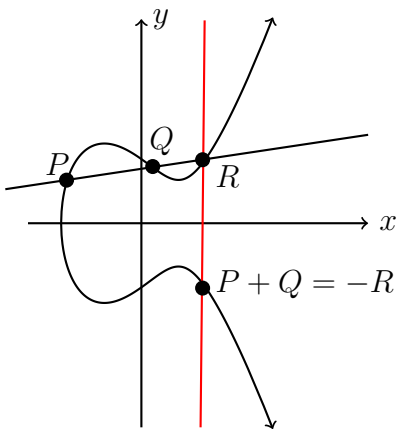
$$y = \left(\frac{y_Q - y_P}{x_Q - x_P}\right)(x - x_P) + y_P.$$

Note that since $x_P \neq x_Q$, the slope of this line is finite. Next, find the unique third point $R = (x_R, y_R)$ on the intersection of this line and the curve given by (3.1), which can be found to have coordinates

$$x_R = \left(\frac{y_Q - y_P}{x_Q - x_P}\right)^2 - a - x_P - x_Q, \text{ and}$$
$$y_R = \left(\frac{y_Q - y_P}{x_Q - x_P}\right)(x_R - x_P) - y_P,$$

noting that a is the coefficient of the x term in (3.1). In this case, $P + Q$ is defined as

$-R = (x_R, -y_R)$, so that its $y-$coordinate is $\left(\frac{y_Q - y_P}{x_Q - x_P}\right)(x_P - x_R) - y_P$.



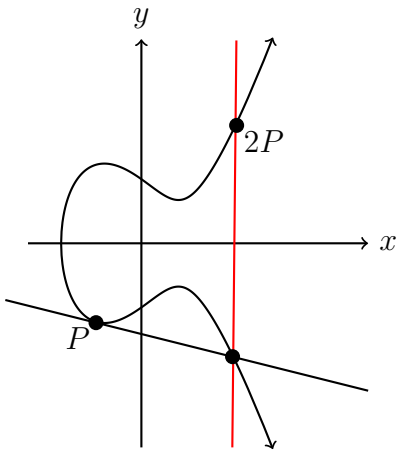
Case 2: $P = Q$.

In this case, we must define $2P = P + P$, where P is a solution to (3.1). The slope of the tangent line to P, $y'(P)$, can be thought of as a type of replacement for the slope of line line between P and Q from Case 1. The value of $y'(P)$ can be found by implicitly differentiating the equation (3.1), and if $y_P \neq 0$, can be found to equal

$$y'(P) = \frac{f'(x_P)}{2y_P} = \frac{3x_P^2 + 2ax_P + b}{2y_P}, \tag{3.2}$$

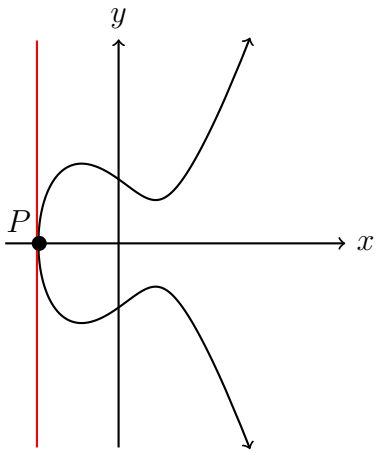
where $f(x) = x^2 + ax + b$ is the right-hand side of (3.1). If $y_P \neq 0$, in an analogous way to Case 1, we define $2P$ to be the additive inverse of the third point on tangent line to P that intersects the curve, which will have coordinates

$$x_{2P} = (y'(P))^2 - a - 2x_P, \text{ and} \tag{3.3}$$
$$y_{2P} = (y'(P))(x_P - x_{2P}) - y_P. \tag{3.4}$$



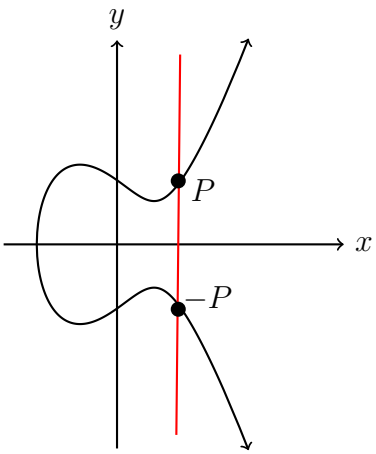
On the other hand, remaining in Case 2, suppose that $y_P = 0$. Since we only consider non-singular curves in our discussion, we require that there be a unique tangent

line to the graph of equation (3.1) at P. Since the slope of this line is $\frac{f'(x_P)}{2y_P}$ by (3.2), we must have that $f'(x_P) \neq 0$, so that the slope is infinite and the tangent line is vertical. In this case, we define $2P$ to be ∞.



Case 3: $P = -P$, but $P \neq Q$.

Notice that in this case, $x_{-P} = x_P$, but $y_{-P} = -y_P$ are distinct. Geometrically, this means that the line passing through these two points is vertical. We define $P + (-P) = \infty$, which must hold because $-P$ is the additive inverse of P in the group.



Elliptic curves modulo a prime

When a, b, and c are integers, the equation (3.1) defining an elliptic curve equation can be considered modulo a prime number p,

$$y^2 \equiv x^3 + ax^2 + bx + c \bmod p. \tag{3.5}$$

There are clearly only finitely many solutions to this equation, which makes solutions very different than solutions over $\mathbb{Q}$ or $\mathbb{R}$.

Let $E(p)$ denote the set of all solutions to (3.5), along with another point denoted "∞." Interestingly, $E(p)$ is a group under the same addition law as for the original elliptic curve, when division by an integer is replaced by a product with the multiplicative inverse of this integer. The slopes and derivatives can be calculated using the equations stated above, but now we do not have a well-defined geometric representation. The elliptic curve $E(p)$ is non-singular if the discriminant Δ of E is not a multiple of p.

The group $E(p)$ is called the *reduction of the elliptic curve E modulo p*. Elliptic curve groups modulo a prime are integral to elliptic curve cryptography, as we begin to see in the next subsection.

Elliptic curve Diffie-Hellman key exchange

Like the traditional version, the Elliptic Curve Diffie-Hellman key exchange provides two parties the ability to agree on a shared private key, but this process uses an elliptic curve modulo a prime. Using this key, like its counterpart, Elliptic Curve ElGamal then enables secret communications over a public channel.

As in the original Diffie-Hellman key exchange, the two parties, Alice and Bob, and first aim to agree on a secret key. In order to do this, both parties first agree on large prime p, and a basepoint G on the the elliptic curve $E(p)$ defined by the equation $y^2 \equiv x^3 + ax^2 + bx + c \bmod p$. This information is public.

Next, Alice and Bob will each select a random positive integer as their private key. Suppose Alice has private key n_A, and Bob has private key n_B. Alice then calculates $n_A G$ (i.e., G added n_A times), and Bob calculates $n_B G$ (G added n_B times), and these two points on $E(p)$ are published publicly.

The private key known only to Alice and Bob is a point on $E(p)$. In order to find this shared key, each party will add the other party's public key (a point of the elliptic curve $E(p)$) to itself the number of times determined by their own private.

More specifically, Alice takes the public key $n_B G \in E(p)$ and adds it to itself n_A times. Likewise, Bob takes the public key $n_A G$, and adds it to itself n_B times. Since

$$n_A(n_B G) \equiv n_B(n_A G) \bmod p.$$

both Alice and Bob know this common point on $E(p)$, and call this shared private key K.

To see how the Elliptic Curve Diffie-Hellman key exchange works in practice, consider the following example.

Example 5. We choose the prime $p = 7177$, so that our elliptic curve E will be defined by the equation

$$y^2 \equiv x^3 + 8 \bmod 7177.$$

Our selected basepoint is $G = (1, 3)$. Once again, the private key for each party is chosen as an integer between 1 to $p - 1 = 7176$. We choose $n_A = 111$ and $n_B = 24$. Using their own private key, each party can then generate their public key: Alice publishes $111G = (6350, 1252)$ and, and Bob publishes $24G = (3705, 5105)$ publicly. Finally, Alice finds $24(111G) = 24(6350, 1252)$ and Bob finds $111(24G) = 111(3705, 5105)$, both resulting in the shared private key $K = (11 \cdot 24)G = 2664G = (500, 4650)$.

Elliptic curve ElGamal encryption

To implement the Elliptic Curve ElGamal Cryptosystem, the Alice and Bob first agree on a shared private key K using the Elliptic Curve Diffie-Hellman key exchange, as described in the previous subsection. Therefore, both Alice or Bob know the prime p, the elliptic curve $E(p)$, and the basepoint G on $E(p)$, which are public data, but are the only parties to know the point K on $E(p)$. Moreover, Alice knows her private key $n_A \in \mathbb{Z}$, and Bob knows his, $n_B \in \mathbb{Z}$.

To begin the encryption, the sender, Alice, should first encode her message into a point $M \in E(p)$. For example, she could break a numerical representation of the message into blocks, which are each less than p. Given a block m, she can then find a point on the curve $E(p)$ with x-coordinate is m, and call this point M.

After translating her message into a point M on $E(p)$, Alice chooses a positive integer z. To encrypt a message to send to Bob, Alice then calculates the following two points on $E(p)$:

$$\begin{aligned} Y_1 &\equiv z(n_A G) \equiv (zn_A)G && \mod p \\ Y_2 &\equiv M + zK && \mod p. \end{aligned}$$

She then sends the pair of points (Y_1, Y_2) to Bob over the public channel.

Using the pair (Y_1, Y_2), Bob can now decrypt by calculating the following:

$$\begin{aligned} Y_2 - n_B Y_1 &\equiv (M + zK) - n_B(zn_A G) && \mod p \\ &\equiv M + zK - zK && \mod p \\ &\equiv M && \mod p. \end{aligned}$$

This process is secure due to the difficulty of the elliptic curve discrete logarithm problem.

Lenstra's elliptic curve factoring algorithm

Lenstra's Elliptic Curve Factoring Algorithm is another factoring method, that in its functionality, is very similar to Pollard's $p-1$ algorithm. However, it is much more powerful: It is considered the third best factoring algorithm currently known, and the best for numbers with prime factors of a certain size.

The goal of Lenstra's Algorithm is to find a non-trivial factor of a composite integer n. The process is as follows:

- Pick an elliptic curve E, $y^2 = x^3 + ax^2 + bx + c$, and coefficients $a, b, c \in \mathbb{Z}$.
- Check that the discriminant Δ of E satisfies

  $$\Delta \not\equiv 0 \mod n.$$

  If $\Delta \equiv 0 \mod n$, replace the curve E with a new elliptic curve until the required condition is satisfied.

- Choose a random point P with integer coordinates on E.
- Successively double the point; i.e., start computing the sequence

$$P, 2P, 4P, 8P, \ldots,$$

  and reduce their coefficients modulo n.
- Since our computations are performed modulo n, if y is the second coordinate of a point in the sequence, to double this point, we must compute the slope that appears in the formula for doubling, which requires finding the multiplicative inverse of $2y$ modulo n. To do so, we use the Extended Euclidean Algorithm.

  However, it is possible that after applying the Euclidean Algorithm to $2y$ and n, we find that $(2y, n) \neq 1$. This can happen since n is not prime, so that the elliptic curve E modulo n is *not* necessarily a group. In this case, we have found a factor of n, namely $(2y, n)$.
- However, if $(2y, n) = n$, we failed to obtain a non-trivial factor of n. In this case, we pick a new point P, or both a new elliptic curve E and point P on E, and repeat the algorithm described thus far.

To see this process in action, consider the following example.

Example 6. Suppose that our goal is to factor $n = 533$. We fix the elliptic curve E given by the equation $y^2 = x^3 + 8$, and we choose the point $P = (1, 3)$ on E.

Next, we calculate the slope needed to find $2P$ modulo 533 using (3.2),

$$3{x_P}^2(2y_P)^{-1} \equiv 3 \cdot 1 \cdot 6^{-1} \equiv 3 \cdot 89 \equiv 267 \bmod 533.$$

From here, using (3.3), we find that $x_{2P} = 2 \cdot 1 - 267^2 \equiv 398 \bmod 533$ and $y_{2P} = 267(1 - 398) - 3 \equiv 65 \bmod 533$, so that $2P = (398, 65)$.

Next, we attempt to find $4P$ modulo n. Using (3.2), the new slope should equal

$$3x_{2P}^2(2y_{2P})^{-1} \equiv 3398^2 \cdot 130^{-1} \bmod 533.$$

However, after applying the Euclidean Algorithm, we find that 130 does not have an inverse modulo 533, since $(130, 533) = 13 \neq 1$. Therefore, we have found a nontrivial factor of 533, namely 13. This gives us the factorization $533 = 13 \cdot 41$.

Note that an analogous algorithm can be performed by successive addition of the point P; i.e., successively computing the points

$$P, 2P, 3P, 4P, 5P,$$

modulo n.

A new elliptic curve public key cryptosystem

One of the goals of our REU was to create our own elliptic curve cryptosystem. In this section, we describe this cryptosystem in detail. You will see that it uses many of the mathematical concepts that we have studied in depth throughout the program, such as the inverse of a unit modulo an integer, the order of an element of a finite group, and elliptic curves modulo a prime.

Inspired by the idea of making an elliptic curve analogue of the RSA cryptosystem, our cryptosystem relies on the choice of two secret large primes p and q, which are distinct. By convention, we assume $p < q$. If $N = pq$, then we make use of an elliptic curve modulo a prime x whose order is a multiple of N. It is possible to find such an elliptic curve using an algorithm described in [1], although we do not describe it here.

Our cryptosystem relies on first selecting a secret basepoint $G \in E(x)$ with order q. We can do this by picking a random point that is not the identity, and then multiplying it by q to see if this yields the identity. If it does, since q is prime and the order of any element is a divisor of the order of the group $E(x)$. If it does not, we must then try another point until we find one with the desired order.

Before proceeding, we state and prove a useful lemma.

Lemma 7. *Fix a prime x and an integer n. Suppose that E is an elliptic curve for which $E(x)$ is non-singular. If G is a point of order n on $E(x)$, then*

$$G, 2G, \ldots, (n-1)G, nG = \infty$$

are distinct points of $E(x)$.

Proof. We proceed by way of contradiction. Suppose that there exist two points, jG and kG, where $1 \leq j < k \leq n$, such that $jG = kG$. Subtracting jG from both sides of this equation, we obtain the equality

$$\infty = jG - jG = kG - jG = (k-j)G;$$

i.e., $\infty = (k-j)G$. However, since $j - k < j < n$, this would imply that the order of G is at most $k - j \leq n - 1$, a contradiction. Thus jG and kG must be distinct points on $E(x)$. □

Encryption and decryption

Recall that we have fixed secret large primes p and q, and a prime x for which $N = pq$ divides $E(x)$. Once we have fixed the basepoint $G \in E(x)$ of order q, we choose a large positive integer $b < q$ and use it to define a "decryption set" as follows:

$$S = \{G, 2G, 3G, \ldots, (b-1)G, bG\}. \tag{4.1}$$

The b points in this set are distinct (and none are ∞) by taking $n = q$ in Lemma 7.

Define the point K on $E(x)$ as

$$K \equiv pG \bmod x,$$

and publish the triple $(K, E(x), b)$ on the public channel. Note that since $p < q$ and G has order q on $E(x)$, $K \neq \infty$ on the curve.

To encrypt a message, the sender first completes the following two steps:

1. Break the message into blocks of size at most b.

2. To encrypt a block m, compute

$$Y \equiv mK \bmod x,$$

and send the point $Y \in E(x)$ across the public channel.

Finally, to decrypt, the receiver must first find the multiplicative inverse ℓ of p modulo x using the Euclidean Algorithm. Next, they multiply the encrypted message, encoded into $Y \in E(x)$, by ℓ, obtaining the point

$$\ell Y \equiv \ell(mK) \equiv (\ell m)(pG) \equiv (\ell p)(mG) \equiv mG \bmod x.$$

The receiver can then find mG among the elements of the decryption list (4.1), and its index in the list will reveal the original plaintext m. Since we require $m \leq b$, we are ensured to find $\ell Y \equiv mG \bmod x$ somewhere on the list. Its position in the list is precisely m, the original message.

Security of the cryptosystem

First, the public data, consisting of the triple $(K, E(x), b)$ and the encrypted message $Y \in E(x)$, does not provide information on p or q except that they are larger than b, and are factors of $|E(x)|$, which is a multiple of N. Hence we can choose a sufficiently large key N to ensure security even if an attacker is able to calculate the order of $E(x)$, since it will be extremely difficult to find the factors p and q from this value. Second, even if such a factorization is found, for an attacker to find the basepoint G, required to generate the decryption list, they must solve the elliptic curve discrete logarithm problem, of which there is no general efficient algorithm.

Python code implementations

The following are SAGE codes were built to implement concepts and algorithms discussed in this paper, and were utilized in our REU program.

Greatest common divisor

```
def gcd(a,b):
    while b:
        a,b = b,a%b
    return a
```

Extended Euclidean Algorithm using Bézout's equation

```
def xgcd(a,b):
    prevx,prevy,x,y = 1,0,0,1
    while b:
        q = a//b
        x,prevx = prevx-q*x,x
        y,prevy = prevy-q*y,y
        a,b = b,a%b
    return a, prevx, prevy
```

Pollard's $p-1$ factoring algorithm

```
def pollard(mod,a):
    from fractions import gcd
    d = gcd(a,mod)
    i = 2
    while i<=100 and d==1:
        a = a**i%mod
        d = gcd(a-1,mod)
        print "a_"+str(i),"=",str(a),"and d_"+str(i), "= "+str(d)
        i+=1
    if d>1:
        print "Pollard's p-1 algorithm found a factor: "+str(d)
    else:
        print "Pollard's p-1 algorithm did not find a factor."
```

Elliptic curve addition group law

```
def ellipticAddMod(coefficients,point1,point2,modulus):
    # Define names that are easier to type.
    a,b,c,m = coefficients[0],coefficients[1],coefficients[2],modulus
    x1,y1,x2,y2 = point1[0]%m,point1[1]%m,point2[0]%m,point2[1]%m
    # Check for a singular curve.
    if (a**2*b**2-4*a**3*c-4*b**3-27*c**2+18*a*b*c)%m == 0:
        # Check if the points are the singular point.
        if xgcd(2*y1,m)[1] == 0 and (3*x1**2 + a*2*x1 + b) == 0:
            return("point1 is singular, try a different point or curve")
        elif xgcd(2*y2,m)[1] == 0 and (3*x2**2 + a*2*x2 + b) == 0:
            return("point2 is singular, try a different point or curve")
    # Escape the preceding if statement.
    else: pass
    # Check if the points are on the curve.
    if (y1**2 % m) <> ((x1**3 + a*x1**2 + b*x1 + c) % m):
        return("point1 not on curve")
    elif (y2**2 % m) <> ((x2**3 + a*x2**2 + b*x2 + c) % m):
        return("point2 not on curve")
    # Implement the case for x1 != x2.
    elif x1 <> x2:
        if gcd(x2-x1,m) == 1:
            A = (y2 - y1)*xgcd((x2-x1),m)[1]
            x3 = (A**2 - a - x1 - x2) % m
            y3 = (A*(x1 - x3) - y1) % m
            return((x3,y3))
```

```
        else:
            return("The 'slope' is not well defined for these points.")
    # Implement the case for point1 = point2.
    elif y1 == y2 and y1 <> 0:
        if gcd(2*y1,m) == 1:
            A = (3*x1**2 + a*2*x1 + b)*xgcd(2*y1, m)[1]
            x3 = (A**2 - a - x1 - x2) % m
            y3 = (A*(x1 - x3) - y1) % m
            return((x3,y3))
        else:
            return("The 'derivative' is undefined at this point")
    # The last possibility should be where point1 = -point2.
    else:
        return("point1 + point2 = the point at infinity")
```

Adding point on elliptic curve to itself multiple times

```
def scalMultiply(sM, coefficients,x,y,p):
    # sM is the number of times added
    base = (x,y)
    # result is the identity to begin with,
    # make sure the add function handles the identity
    # a concise way to add point to itself n times
    if(sM % 2 == 1):
        result = base
    sM = sM >> 1
    base = ellipticAddMod(coefficients,base, base,p)
    while (sM > 0):
        if (sM % 2 == 1):
                result = ellipticAddMod(coefficients,result, base,p)
        sM = sM >> 1
        base = ellipticAddMod(coefficients,base, base,p)
    return result
```

Lenstra's elliptic curve factoring method

```
def ellipticFactor(point,coeffs,modulus,maxsteps,style):
    prev = point
    r = ellipticAddMod((0,coeffs[0],coeffs[1]),point,point,modulus)
    s = 1
    if style == "a":
        while type(r) <> type(str()) and s < maxsteps:
            prev = r
            r = ellipticAddMod((0,coeffs[0],coeffs[1]),point,r,modulus)
            # step iterator
            s += 1
        if gcd(prev[0] - point[0],modulus) not in [modulus, 1] :
            return(gcd(prev[0]-point[0],modulus))
        else:
            return("No factor found.")
    elif style == "d":
        while type(r) <> type(str()) and s < maxsteps:
            prev = r
```

```
            r = ellipticAddMod((0,coeffs[0],coeffs[1]),r,r,modulus)
            # step iterator
            s += 1
            #checks if string
        if gcd(2*prev[1],modulus) not in [modulus,1] :
            return(gcd(2*prev[1],modulus))
        else:
            return("No factor found.")
    else:
        return("Must input \"d\" (doubling) or \"a\" (adding).")
```

Generate points on a given elliptic curve

```
def ellipticTestPoints(threeCoeffs,modulus,xMax,yMax):
    xdict = {}
    ydict = {}
    pointlist = []
    # The dictionary would account for any sort of duplicate.
    for i in xrange(1,xMax+1):
        xdict[(i**3 + threeCoeffs[0]*i**2 + threeCoeffs[1]*i
              + threeCoeffs[2])%modulus] = i
    for j in xrange(1,yMax+1):
        if (j**2)%modulus in xdict.keys():
            pointlist.append((int(xdict[(j**2)%modulus]),j))
    return pointlist
```

Acknowledgements

The REU in which this manuscript was developed was funded by National Science Foundation Grant DMS #1501404/1623035 through the University of Kansas. The REU was hosted at the University of Texas at El Paso (UTEP) under the direction of Prof. Emily Witt and Prof. Daniel Hernández, assisted by Angel Agüero. We thank UTEP for their generous hospitality during the duration of the REU. We also thank Gordan Savin for allowing us to use his notes as a reference.

Finally, a special thanks to Jack Jeffries, a very cool guy that couldn't make it.

References

[1] D. Boneh, K. Rubin, and A. Silverberg. Finding composite order ordinary elliptic curves using the Cocks-Pinch method. *Journal of Number Theory*, 131(5):832 – 841, 2011. Elliptic Curve Cryptography.

[2] Gordan Savin. Numbers, groups and cryptography.

[3] Wade Trappe and Lawrence C Washington. *Introduction to cryptography with coding theory*. Pearson Education India, 2006.