Applications Project: Elliptic Curve Cryptography

Carl Harris
Math 31: Topics in Algebra
Dartmouth College

December 1, 2020

1 Introduction and Historical Background

1.1 Paper Organization

In this paper, I provide an introduction to the history, theory, and applications of elliptic curve cryptography (ECC). In the first section, "Introduction and Historical Background," I provide a concise overview of public key cryptography and the advent of elliptic curve cryptography. In the second section, "Background on Elliptic Curves," I introduce elliptic curves over finite fields, which are the basis for ECC implementations. In Section 3, "Elliptic Curve Subgroups," I describe the generation of elliptic curve subgroups and the properties relevant to my ECC implementation example. In Section 4, "Elliptic Curve Cryptosystems: Theory and MATLAB Implementation" I provide an overview of how cryptographic systems are constructed from elliptic curves and used to encrypt and decrypt messages, along with a MATLAB implementation of this process.

1.2 Overview of Cryptography

Cryptography is a science dating thousands of years. The ability to send and receive messages securely has long been critical to communication. For example, in 60 BCE Julius Caesar used a relatively simple cipher to send messages to those in battle [5]. A perpetual challenge for cryptographers prior to the 1970s was that cryptographic systems relied on the secure exchange of the decryption method. For example, in order for Julius Caesar to send a secure message, the intended recipient would need to know (or be in possession of) the "key" to decrypt it. This represents a fundamental vulnerability for pre-1970s encryption methods: decryption keys always have the potential to be intercepted. But, these methods relied on the assumption that decryption keys can be transferred securely. If this cannot be accomplished (as is often the case), encryption becomes far less secure in ensuring private communication [13].

1.3 Public Key Cryptography

In 1977, a breakthrough in cryptography emerged. Rather than relying on the secure transfer of secret keys, Whitfield Diffie and Martin Hellmen created a system with separate encryption and decryption keys. The process used to encrypt data could be made public (the "public key"), so messages could be encrypted by others with access to the public key and sent to the desired party. A different process (the "private key") would be used to decrypt incoming encrypted messages. The private key is kept secret. So, while anyone with access to the public key can encrypt messages, only the intended recipient is able to decrypt them, because only he or she has access to the private key. If encrypted messages were to be intercepted by a "man in the middle" (i.e. not the intended recipient), they would be unencryptable, because the man in the middle would lack access to the secret private key necessary to do so. Termed an "asymmetric key exchange," this system marked the dawn of modern cryptography [13].

Such encryption schemes rely on a "trapdoor function." That is, a function that is "easy" to perform in one direction (to create a private and public key) but very difficult to perform in the other (i.e. finding the private key from the public key). In the first public key cryptosystem, this was implemented via the RSA algorithm. The RSA system uses operations with prime numbers as its trapdoor function. Multiplying two large prime numbers is relatively easy. But factoring the product of two primes into its component primes is much more difficult. This makes prime multiplication and associated factorization an acceptable process to construct a public key cryptosystem [13].

1.4 Advent of Elliptic Curve Cryptography

One of the drawbacks of the RSA algorithm, however, is that it relies on the use of very large prime numbers, which can take relatively large amounts of storage space on digital devices. Large prime numbers are required to make factoring their products sufficiently difficult, which is essential to encryption security. A further drawback of RSA is that there are factoring algorithms that greatly reduce the time required to "guess-and-check" the component primes relative to a naive approach. For this reason, as computational power grows exponentially and factorization algorithms become more efficient, so too does the size of large numbers required to secure data, leading to increasing storage requirements [6].

As an alternative, elliptic curve cryptography (ECC) is a cryptosystem that requires far smaller key sizes for the same level of security. Introduced independently by Koblitz and Miller in 1985, ECC proposes using an elliptic curve over a finite field to encrypt and decrypt data. Elliptic curves are introduced in the next section, and have proven highly effective in ushering the next era of encryption [10].

2 Background on Elliptic Curves

2.1 Definition of an Elliptic Curve

In the context of elliptic curve cryptography, consider the finite field of integers modulo p, where p is a prime number. This is denoted $\mathbb{Z}_p$. Then, the elliptic curve, E, over $\mathbb{Z}_p$, is defined as follows:

$$E := \{(x, y) \in (\mathbb{Z}_p)^2 \mid y^2 \equiv x^3 + ax + b \bmod p, 4x^3 + 27b^2 \not\equiv 0 \bmod p\} \cup \{0\},$$

where the identity element is the "point at infinity," or the limiting point of the curve (an explanation of which is beyond the scope of this paper, but can be found in [11, 12]) and $a, b \in \mathbb{Z}_p$. It can be shown that the elliptic curve over $\mathbb{Z}_p$ is an abelian group [2].

With this definition in mind, we then consider two important operations on E: point addition and scalar multiplication.

2.2 Point Addition

Here, we must first define elements $P = (x_1, y_1), Q = (x_2, y_2) \in E$, where $p \neq 2, 3$. For $p = 2, 3$, a different set of operations is required, which I don't include here because in all examples provided here (and practical cryptosystems), $p > 3$. Then $P + Q = (x_3, y_3) \in E$, where:

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1. \end{aligned}$$

If $P = Q$, then $\lambda = (3x_1^2 + a)(2y_1)^{-1} \bmod p$. If $P \neq Q$, then $\lambda = (y_2 - y_1)(x_2 - x_1)^{-1} \bmod p$ [2,10]. Here, $^{-1}$ denotes the multiplicative modular inverse. In this context, it means that for a given element, x, the modular multiplicative inverse is an element, $x^{-1} \in \mathbb{Z}_p$, such that $xx^{-1} \bmod p = 1$ [2]. The multiplicative inverse can be found using Bezout's identity, as detailed in [9] (and implemented in the provided MATLAB function `ModMultInverse`).

As an example, first consider the addition of points $P = (0, 1)$ and $Q = (0, 1)$ on the curve $y^2 = x^3 + x + 1 \bmod 7$. Since $P = Q$, we have:

$$\begin{aligned} \lambda &= (3x_1^2 + a)(2y_1)^{-1} \bmod p \\ &= (3 \cdot 0 + 1)(4) \bmod 7 \\ &= 4 \bmod 7 \\ &= 4. \end{aligned}$$

Then substituting $\lambda = 4$, we find:

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \bmod 7 \\ &= 4^2 - 0 - 0 \bmod 7 \\ &= 2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \bmod 7 \\ &= \lambda(0 - 2) - 1 \bmod 7 \end{aligned}$$

$$= 5.$$

Therefore, $(0,1) + (0,1) = (2,5)$ on this curve.

Second, consider an example where $P = (0,1)$ and $Q = (2,2)$ on the same curve. Since $P \neq Q$, we have:

$$\begin{aligned}\lambda &= (y_2 - y_1)(x_2 - x_1)^{-1} \bmod p \\ &= (2-1)(2-0)^{-1} \bmod 7 \\ &= 1 \cdot 4 \bmod 7 \\ &= 4\end{aligned}$$

Then, substituting $\lambda = 4$:

$$\begin{aligned}x_3 &= \lambda^2 - x_1 - x_2 \bmod 7 \\ &= 4^2 - 0 - 2 \bmod 7 \\ &= 14 \bmod 7 \\ &= 0 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \bmod 7 \\ &= 4(0-14) - 1 \bmod 7 \\ &= 6\end{aligned}$$

Therefore, $(0,1) + (2,3) = (0,6)$ on this curve.

2.3 Point Multiplication

For a point $P \in E$, scalar multiplication by $n \in \mathbb{Z}_{>0}$ is defined as follows:

$$nP = \underbrace{P + P + ... + P}_{n \text{ times}}.$$

That is, the scalar multiplication of point P by n is P added to itself n times. If $n = 0$, then $nP = 0$ [2,3].

2.4 Application to Cryptography

In elliptic curve cryptosystems, the "trapdoor" function is based on the computation of a point Q, from the multiplication of another point, P, by a scalar n. The "easy" part of the function is calculating Q given n and P. This operation can be completed relatively quickly. The "hard" function is determining n given Q and P. Known as the "discrete logarithm problem," this is a much more complex task, and no known algorithms can significantly decrease the number of expected operations required to find n [2,3].

3 Elliptic Curve Subgroups

3.1 Cyclic Subgroups

Having established the basic elliptic curve group and associated operations, we then consider subgroups of E. For any point $P \in E$, $\langle P \rangle$ forms a cyclic subgroup [10,11]. The point P is called the "base point" of the subgroup $\langle P \rangle$. For example, consider the curve $y^2 = x^3 - x + 3 \mod 37$ and point $(2,3)$. We can use the function `GenerateSubgroupElements` to generate the elements in the subgroup:

```
GenerateSubgroupElements(Subgroup, EllipticCurve);
```

The output of `GenerateSubgroupElements` is shown below.

```
On the elliptic curve y^2 = x^3 + -1x+ 3 mod 37...
Base point (2, 3), generates the following subgroup:
    0P = 0(2, 3) = 0
    1P = 1(2, 3) = (2, 3)
    2P = 2(2, 3) = (23, 14)
    3P = 3(2, 3) = (21, 17)
```

```
4P = 4(2, 3) = (21, 20)
5P = 5(2, 3) = (23, 23)
6P = 6(2, 3) = (2, 34)
7P = 7(2, 3) = 0
```

In this example, the subgroup $\langle(2,3)\rangle$ on the curve $y^2 = x^3 - x + 3 \mod 37$ has seven elements ($\{0, (2,3), (23,14), (21,17), (21,20), (23,23), (2,34)\}$).

3.2 Subgroup Order

Critical to an effective implementation of ECC is selection of a subgroup with a large number of elements. Finding the order of the subgroup of an elliptic curve with a given base point can be time consuming, so prior knowledge that improves algorithmic efficiency is highly valuable. One such prior is the fact that the order of a subgroup of an elliptic curve is a divisor of the order of the elliptic curve. This means that one only needs to test scalar multiplication of a given base point with the divisors of the elliptic curve's order (from lowest to highest) to determine the order of the subgroup [2]. This is proven below.

Proof: Here, we hope to show that the number of elements n, in the cyclic subgroup generated by P, is a multiple of the larger elliptic curve group's order, N. To do so, let H denote the subgroup generated by P (i.e. $H = \langle P \rangle$), and let G denote the larger elliptic curve group. Since G is a finite group (an elliptic curve over a finite field is a finite group), and H is a subgroup of G, by Lagrange's Theorem [1], the order of G is a multiple order the order of H. Therefore, there is an $n \in \mathbb{Z}$ such that $n|H| = |G|$, so n and $|H|$ must be divisors of $|G|$. Therefore, $|H|$ is a divisor of $|G|$, so the order of H is a divisor of the order of G. This means one only needs to test whether the divisors of $|G|$ are the order of $|H|$. □

Below is an implementation of a function `GenerateDivisors`, that tests all the divisors, D of the order of an elliptic curve to determine the order of the subgroup, from lowest to highest value. The curve selected in this example is $y^2 = x^3 - x + 3 \mod 37$ and the subgroup generator is $(2,3)$. This curve has order 42 [2], which was determined via Schoof's algorithm, an explanation and implementation of which is beyond the scope of this paper (see [8, 14] for details).

To determine the order of the subgroup, this function tests every divisor $d \in D$, from lowest to highest, until it finds that $d(2,3) = 0$ (though I show scalar multiplication with every divisor in this example). The first value for which this occurs is the order of the subgroup (in this case, 7). As expected, every divisor greater than the order is also a divisor of the order (i.e. $14/7 = 2$ and $28/7 = 4$), and therefore when the base point is multiplied with it, the result is zero.

```
EllipticCurve = struct;
EllipticCurve.a = -1;
EllipticCurve.b = 3;
EllipticCurve.p = 37;
EllipticCurve.Order = 42;
Subgroup.BasePoint = [2 3];

GenerateDivisors(Subgroup, EllipticCurve);
```

The output of `GenerateDivisors` is printed below. In a true implementation of this process, the function would quit after determining $7(2,3)$ is the lowest value of n for which $n(2,3) = 0$, and therefore its order.

```
The divisors of 42 are: 1, 2, 3, 6, 7, 14, 21, and 42.
The results of their scalar multiplication is:

1(2, 3) = (2, 3)
2(2, 3) = (23, 14)
3(2, 3) = (21, 17)
6(2, 3) = (2, 34)
7(2, 3) = 0
14(2, 3) = 0
21(2, 3) = 0
42(2, 3) = 0

The lowest value, n, for which n(2, 3) = 0 is 7.
```

```
Therefore, the order of the subgroup generated by (2, 3) is 7.
```

As verification that 7 is the lowest integer for which $n(2,3) = 0$, refer back to the function `GenerateSubgroupElements` in Section 3.1, which prints all the integers from 0 to 7 multiplied by $(2,3)$ to see that 7 is, in fact, the lowest number for which scalar multiplication is 0. In practice, testing divisors rather than every scalar significantly improves computation time, because non-divisor integers (i.e. 4 and 5 in this example) do not need to be tested.

4 Elliptic Curve Cryptosystems: Theory and MATLAB Implementation

4.1 Implementation Overview

Here, I present an example implementation of an elliptic curve cryptosystem. In Section 4.2, I generate an elliptic curve and a cyclic subgroup. I then use this subgroup in Section 4.3 to generate public and private keys for two example agents, Alice and Bob, where Bob is attempting to send a secure message to Alice. In Section 4.4, Bob uses Alice's public key to encrypt a secret message, which he then sends to Alice. In Section 4.5, Alice decrypts this message using her private key.

4.2 Generating an Elliptic Curve and a Subgroup

Here, we begin with the elliptic curve $y^2 = x^3 + x + 1 \mod 463$ (where 463 is the 90th prime number, output by MATLAB's `nthprime(primeNo)` function, where `primeNo` is 90). Using this curve, we then consider the subgroup generated by base point $(0,1)$. Because this is a relatively simple curve and a small subgroup, we are able to manually generate all the elements of the subgroup (to better demonstrate the encryption process). To do so, the function `FindSubgroupOrder` continually multiplies the base point by itself until it reaches the point at infinity, when it terminates. The elements of the subgroup are then all the elements generated before this point, plus the zero element (i.e. the point at infinity), and its order is the number of elements in the subgroup [10].

The code below demonstrates how `FindSubgroupOrder` is used to generate the order and elements in the subgroup `Subgroup` of elliptic curve `EllipticCurve`.

```
EllipticCurve = struct;
primeNo = 90;
EllipticCurve.a = 1;
EllipticCurve.b = 1;
EllipticCurve.p = nthprime(primeNo);
Subgroup.BasePoint = [0 1];

[order, elements] = FindSubgroupOrder(Subgroup, EllipticCurve);
Subgroup.Order = order;
PrintEllipticCurveSummary(primeNo, Subgroup, EllipticCurve)
```

The output of this function is summarized by `PrintEllipticCurveSummary`, which prints the result below.

```
(STEP 1) Encryption using eliptic curve y^2 = x^3 + 1x+ 1 mod 463
    ... Where 463 is the 90th prime number
    ... And base point (0, 1) generates a subgroup of order 237
```

After generating the elliptic curve and subgroup generated by $(0,1)$, we can create public and private keys.

4.3 Public and Private Key Creation and the Key Exchange

Here, we utilize an ECC algorithm to generate public and private keys on the subgroup generated by $(0,1)$ for two example agents, Alice and Bob.

To begin, the algorithm defines Q to be a publicly known point (in this case, I selected the base point $(0,1)$). Alice chooses a secret random integer k_a less than the order of the subgroup (237), and computes k_aQ. Alice's private key is k_a, and her public key is k_aQ. Likewise, Bob chooses a secret integer, k_b, which is his private key. He then computes k_bQ, his public key [10].

Here, the function `GenerateKeyPair` performs this. It selects a random value less than the order of the subgroup and multiplies the base point $(0,1)$ (although it could be any point in the subgroup) by that random value, to generate the public key.

```
[alicePrivateKey, alicePublicKey] = GenerateKeyPair(Subgroup, EllipticCurve);
[bobPrivateKey, bobPublicKey] = GenerateKeyPair(Subgroup, EllipticCurve);
PrintPublicPrivateKeys(alicePrivateKey, alicePublicKey, bobPrivateKey, bobPublicKey)
```

After the public and private keys have been generated, the function `PrintPublicPrivateKeys` generates the output below.

```
(STEP 2) Creating Public and Private Keys...
    Alice's Secret Key (k_a):  24
    Alice's Public Key (k_a * Q): (323, 110)
    Bob's Secret Key (k_b):  66
    Bob's Public Key (k_b * Q): (425, 240)
```

After creating these public and private keys, both Alice and Bob can generate a "common key," $P = k_a k_b Q$. Below is the function for calculating P from a public key and private key. Alice computes P by multiplying Bob's public key ($k_b Q$) by her private key k_a. The function `GenerateCommonKey(bobPublicKey, alicePrivateKey, Subgroup, EllipticCurve)` performs this operation. Similarly, Bob computes P by multiplying Alice's public key ($k_a Q$) by his private key, k_b [10]. This is accomplished using the function `GenerateCommonKey(alicePublicKey, bobPrivateKey, Subgroup, EllipticCurve)` below.

```
aliceCommonKey = GenerateCommonKey(bobPublicKey, alicePrivateKey, Subgroup, EllipticCurve);
bobCommonKey = GenerateCommonKey(alicePublicKey, bobPrivateKey, Subgroup, EllipticCurve);
PrintCommonKeys(aliceCommonKey, bobCommonKey)
```

The common keys generated by Alice and Bob are printed below using `PrintCommonKeys`. Notice that both Alice and Bob generated the same common key, despite using different public/private key combinations, as expected.

```
(STEP 3) Generating Common Keys...
    Alice's Common Key (P = k_a * k_b * Q): ( 61,  45)
    Bob's Common Key (P = k_b * k_a * Q): ( 61,  45)
```

The strength of ECC is derived from the difficulty of determining P . A man in the middle would need to determine P only knowing Q, $k_a Q$, and $k_b Q$, but without knowing k_a or k_b. On complex curves, this process becomes too computationally expensive to be plausible, rendering decryption effectively impossible [10].

4.4 Message Encryption

After generating public and private keys, Alice and Bob can securely encrypt and decrypt messages. The protocol for sending secure messages is as follows:

1. Suppose a message, M has been embedded in the elliptic curve in an agreed upon way (i.e. $M \in E$) and that Bob wants to send a message to Alice and keys have already been exchanged as above. Since the key exchange has occurred, Bob has Alice's public key, $k_a Q$.
2. Bob then chooses a secret random integer, l, and computes the pair of points lQ and $M + l(k_a Q)$ (i.e. the first point is the random integer multiplied by the base point and second is the the message added to the random integer multiplied by Alice's public key).
3. Then, to Bob sends the encrypted message embedded in the pair of points lQ and $M + l(k_a Q)$.
4. After recieving the encrypted message, Alice can multiply the first point by her private key, to generate the point $k_a lQ$. Then, because the subgroup is abelian, $k_a lQ = lk_a Q = l(k_a Q)$, so she can subtract this result from the second point to arrive at $M + l(k_a Q) - k_a(lQ) = M + l(k_a Q) - l(k_a Q) = M$. This is the decrypted message Bob encrypted [10].

To implement this protocol, I let M be a letter in the message, and encrypted every letter individually using its unicode representation. I set the point M to be $(\texttt{unicode}, 0)$, where `unicode` is the unicode representation. For example, the letter `a` is 97 in uniode, so if `a` was a letter in the message to be encrypted I would represent the corresponding M as the point $(97, 0)$. In practice, this is an often faulty assumption, because it assumes $(\texttt{unicode}, 0)$ is an element in the subgroup, which it may not be. In this case, the point $M + l(k_a Q)$ would not necessarily be in the subgroup. However, this program was designed to work for curves of varying complexity, so it is difficult to create an embedding system that works consistently across different curves and subgroups.

Below is the encoding of the message "Hello world," where each letter is encoded as its own message. For each letter the following process was conducted:

1. Select a random integer, l, and multiply the base point l times to generate the first point, lQ.
2. Convert the letter to unicode, and represent the letter point as (`unicode letter`, 0).
3. Multiply the public key l times, to generate $l(k_aQ)$.
4. Add the letter point to $l(k_aQ)$ to generate the second point, $M + l(k_aQ)$ [10].
5. Save each pair of points, corresponding to each letter in the message.

The function `EncryptMessage` performs this process.

```
message = 'Hello world';
Encryption = EncryptMessage(message, Subgroup, alicePublicKey, EllipticCurve);
PrintEncryptionProcess(message, alicePublicKey, Encryption)
```

The results of message encryption are shown below using the `PrintEncryptionProcess` function.

```
(STEP 4) Encrypting message "Hello world" using Alice's public encryption key:
         k_a * Q= (323, 110) generates the following pairs of points
         (one set of points, {(l * Q), (M + l(k_a * Q))}, per letter):
    1. 'H' (Unicode  72), l=130 -> {(130(0, 1)), (( 72, 0) + 130(323, 110))}
        = {(449,  22), (353, 433)}
    2. 'e' (Unicode 101), l=226 -> {(226(0, 1)), ((101, 0) + 226(323, 110))}
        = {(176, 235), (457, 308)}
    3. 'l' (Unicode 108), l=228 -> {(228(0, 1)), ((108, 0) + 228(323, 110))}
        = {( 70, 377), (414, 135)}
    4. 'l' (Unicode 108), l= 38 -> {( 38(0, 1)), ((108, 0) +  38(323, 110))}
        = {(328,  88), (258, 365)}
    5. 'o' (Unicode 111), l=230 -> {(230(0, 1)), ((111, 0) + 230(323, 110))}
        = {( 43, 434), (374,  21)}
    6. ' ' (Unicode  32), l=226 -> {(226(0, 1)), (( 32, 0) + 226(323, 110))}
        = {(176, 235), (388, 308)}
    7. 'w' (Unicode 119), l=115 -> {(115(0, 1)), ((119, 0) + 115(323, 110))}
        = {(294, 115), (527, 182)}
    8. 'o' (Unicode 111), l=189 -> {(189(0, 1)), ((111, 0) + 189(323, 110))}
        = {(280, 110), (259, 362)}
    9. 'r' (Unicode 114), l= 34 -> {( 34(0, 1)), ((114, 0) +  34(323, 110))}
        = {(106, 192), (553, 351)}
    10. 'l' (Unicode 108), l=100 -> {(100(0, 1)), ((108, 0) + 100(323, 110))}
        = {( 35,  93), (462, 356)}
    11. 'd' (Unicode 100), l=217 -> {(217(0, 1)), ((100, 0) + 217(323, 110))}
        = {(290, 206), (172, 148)}
```

4.5 Message Decryption

To decrypt the message, Alice first accepts the encrypted message in the form of the points $\{(499, 22), (353, 433)\}...\{(290, 206), (172,$ which are stored in the `Encryption` structure. For each set of points, $\{lQ, M + l(k_aQ)\}$, corresponding to an encrypted letter, she performs the following:

1. Multiply the first point, lQ, by the private key, k_a, to generate the point k_alQ.
2. Subtract this point, k_alQ, from the second point in the set, $M + l(k_aQ)$ to find M [10].
3. Because M is in the form (`unicode letter`, 0), discard the y coordinate and only consider `unicode letter`.
4. Convert `unicode letter` to text.

Once all the points have been decrypted, the final fullly decrypted message is returned. This process is performed via the `DecryptMessage` function.

```
Decryption = DecryptMessage(Encryption, Subgroup, alicePrivateKey, EllipticCurve);
PrintDecryptionProcess(alicePrivateKey, Decryption, Encryption)
```

The results of the decryption and the decryption process are printed via the `PrintDecryptionProcess` function below.

```
(STEP 5) Decrypting message using Alice's private key, k_a = 24...
    1.  k_a(l * Q)  =  24(449,  22)  = (281, 433)
        (M + l(k_a * Q)) - k(l * Q)  = (353, 433) - (281, 433)  = ( 72,   0) -> 'H'
    2.  k_a(l * Q)  =  24(176, 235)  = (356, 308)
        (M + l(k_a * Q)) - k(l * Q)  = (457, 308) - (356, 308)  = ( 72,   0) -> 'e'
    3.  k_a(l * Q)  =  24( 70, 377)  = (306, 135)
        (M + l(k_a * Q)) - k(l * Q)  = (414, 135) - (306, 135)  = ( 72,   0) -> 'l'
    4.  k_a(l * Q)  =  24(328,  88)  = (150, 365)
        (M + l(k_a * Q)) - k(l * Q)  = (258, 365) - (150, 365)  = ( 72,   0) -> 'l'
    5.  k_a(l * Q)  =  24( 43, 434)  = (263,  21)
        (M + l(k_a * Q)) - k(l * Q)  = (374,  21) - (263,  21)  = ( 72,   0) -> 'o'
    6.  k_a(l * Q)  =  24(176, 235)  = (356, 308)
        (M + l(k_a * Q)) - k(l * Q)  = (388, 308) - (356, 308)  = ( 72,   0) -> ' '
    7.  k_a(l * Q)  =  24(294, 115)  = (408, 182)
        (M + l(k_a * Q)) - k(l * Q)  = (527, 182) - (408, 182)  = ( 72,   0) -> 'w'
    8.  k_a(l * Q)  =  24(280, 110)  = (148, 362)
        (M + l(k_a * Q)) - k(l * Q)  = (259, 362) - (148, 362)  = ( 72,   0) -> 'o'
    9.  k_a(l * Q)  =  24(106, 192)  = (439, 351)
        (M + l(k_a * Q)) - k(l * Q)  = (553, 351) - (439, 351)  = ( 72,   0) -> 'r'
    10. k_a(l * Q)  =  24( 35,  93)  = (354, 356)
        (M + l(k_a * Q)) - k(l * Q)  = (462, 356) - (354, 356)  = ( 72,   0) -> 'l'
    11. k_a(l * Q)  =  24(290, 206)  = ( 72, 148)
        (M + l(k_a * Q)) - k(l * Q)  = (172, 148) - ( 72, 148)  = ( 72,   0) -> 'd'

 Fully decrypted message: Hello world
```

5 Conclusion

This paper provides a brief overview of the history, theory, and practical implementation of ECC. The MATLAB code for generating subgroup order and elements corresponding to the examples in Section 3 can be found in Appendix A. The MATLAB code for key creation and message encryption and decryption, as demonstrated in Section 4, can be found in Appendix B.

References

[1] Allen, Samantha. “Cosets and Lagrange’s Theorem (Lecture 11).” Math 31: Topics in Algebra.

[2] Corbellini, Andrea. “Elliptic Curves Over Finite Fields and the Discrete Logarithm.” Elliptic Curve Cryptography: A Gentle Introduction, 23 May 2015, andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/.

[3] Corbellini, Andrea. “Elliptic Curves Over Real Numbers and the Group Law.” Elliptic Curve Cryptography: A Gentle Introduction, 17 May 2015, andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/.

[4] Corbellini, Andrea. “Key Pair Generation and Two ECC: Algorithms: ECDH and ECDSA.” Elliptic Curve Cryptography: A Gentle Introduction, 30 May 2015, andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/.

[5] d’Agapeyeff, Alexander. Codes and ciphers-A history of cryptography. Read Books Ltd, 2016.

[6] Gaitonde, Animesh. “Introduction to Elliptic Curve Cryptography.” Medium, Dev Genius, 5 June 2020, medium.com/dev-genius/introduction-to-elliptic-curve-cryptography-567e47b0e49e.

[7] Kapoor, Vivek, Vivek Sonny Abraham, and Ramesh Singh. ”Elliptic curve cryptography.” Ubiquity 2008.May (2008): 1-8.

[8] Kinakh, Iaroslav, and Ihor Iakymenko. ”Reliability of Schoof algorithm and its computational complexity.” 2009 10th International Conference-The Experience of Designing and Application of CAD Systems in Microelectronics. IEEE, 2009.

[9] Karst, Nathan. Modular Multiplicative Inverses in Matlab. 30 Jan. 2014, www.nathankarst.com/blog/modular-multiplicative-inverses-in-matlab.

[10] Koblitz, Neal, Alfred Menezes, and Scott Vanstone. ”The state of elliptic curve cryptography.” Designs, codes and cryptography 19.2-3 (2000): 173-193.

[11] Koblitz, Neal. ”Elliptic curve cryptosystems.” Mathematics of computation 48.177 (1987): 203-209.

[12] Sathyanarayana, S. V., M. Aswatha Kumar, and KN Hari Bhat. ”Symmetric Key Image Encryption Scheme with Key Sequences Derived from Random Sequence of Cyclic Elliptic Curve Points.” IJ Network Security 12.3 (2011): 137-150.

[13] Sullivan, Nick. “A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography.” The Cloudflare Blog, Cloudflare, 23 Sept. 2013, blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/.

[14] Qun-ying, L. I. A. O. ”Some Sufficient and Necessary Conditions for Primitive Elements over Finite Fields [J].” Journal of Sichuan Normal University (Natural Science) 2 (2005).

A Appendix A: Subgroup Order and Elements (MATLAB Code)

Below is the code to generate the results from **Section 3. Elliptic Curve Subgroups**.

A.1 Application Driver: SubgroupMain.m

```
%% PART 1: CREATE AN ELLIPTIC CURVE
% Create an elliptic curve in the form y^2 = x^3 + ax + b mod p, where
% a is 'EllipticCurve.a', b is 'EllipticCurve.b', and p is the 'primeNo'th
% prime number. The default curve is y^2 = x^3 + -x + 3 mod 37.
EllipticCurve = struct;
EllipticCurve.a = -1;
EllipticCurve.b = 3;
EllipticCurve.p = 37;
EllipticCurve.Order = 42;

%% PART 2: TEST ELLIPTIC CURVE DIVISORS TO FIND SUBGROUP ORDER
% Using the point, p = 'Subgroup.BasePoint', test each divisor, d, of
% 'EllipticCurve.Order' to see if pd = 0. The lowest d for which this occurs
% is the order of the subgroup. The default base point for
% y^2 = x^3 + -x + 3 mod 37 is (2,3).
Subgroup.BasePoint = [2 3];
TestDivisors(Subgroup, EllipticCurve);

%% PART 3: RETURN THE ELEMENTS IN THE SUBGROUP
% Do a naive search and return all the elements generated by the base
% point and the order of the subgroup generated by the base point (this
% implementation for finding the subgroup order is very inefficient, see
% Schoof's algorithm for a feasible implementation for more complex curves).
GenerateSubgroupElements(Subgroup, EllipticCurve);
```

A.2 Application Driver: TestDivisors.m

```
% Find the divisors of the elliptic curve's order and test them sequentially
% to see if n(Subgroup.BasePoint) = 0 and output the results. If
% n(Subgroup.BasePoint), record the lowest value of n for which this is true
% and note this is the order of the subgroup generated by the base point.
function TestDivisors(Subgroup, EllipticCurve)
orderDivisors = divisors(EllipticCurve.Order);

fprintf('The divisors of %0.0f are:', EllipticCurve.Order);
for i = 1:length(orderDivisors)-1
    fprintf(' %0.0f,', orderDivisors(i));
end
fprintf(' and %0.0f.\n', orderDivisors(end))

fprintf('The results of their scalar multiplication is: \n\n')
lowestDivisor = nan;
for i = 1:length(orderDivisors)
    n = orderDivisors(i);
    basePoint = Subgroup.BasePoint;
    p = Subgroup.BasePoint;

    [p, isZero] = IsZeroMultiplication(basePoint, p, n, EllipticCurve);

    if isZero
        fprintf('%0.0f(%0.0f, %0.0f) = %0.0f\n', n, basePoint(1), basePoint(2), p);
        if isnan(lowestDivisor)
            lowestDivisor = n;
        end
        break
    else
        fprintf('%0.0f(%0.0f, %0.0f) = (%0.0f, %0.0f)\n', n, basePoint(1), ...
                basePoint(2), p(1), p(2));
    end
end

fprintf('\nThe lowest value, n, for which n(%0.0f, %0.0f) = 0 is %0.0f.', ...
    basePoint(1), basePoint(2), lowestDivisor)
fprintf(['\nTherefore, the order of the subgroup generated by ',...
         '(%0.0f, %0.0f) is %0.0f\n'], basePoint(1), basePoint(2), ...
```

```
        lowestDivisor)
end
```

A.3 Application Driver: IsZeroMultiplication.m

```
% Scalar multiplication with boolean for whether the zero element has been
% reached.
function [p, isZero] = IsZeroMultiplication(basePoint, p, n, EllipticCurve)
i = 1;
isZero = false;
while i < n
    p = AddPoints(p, basePoint, EllipticCurve);
    i = i+1;
    if isnan(p)
        p = 0;
        isZero = true;
        break
    end
end
end
```

A.4 Application Driver: GenerateSubgroupElements.m

```

% Find the order and all the elements generated by base point 'q' on curve
% 'curve'.
function GenerateSubgroupElements(Subgroup, EllipticCurve)

% Create the 'elements' array which will hold all the elements in the
% subgroup and set the initial 'currPoint' value to the base point
elements = zeros(0,2);
currPoint = Subgroup.BasePoint;

fprintf(['\nThe eliptic curve y^2 = x^3 + %0.0fx',...
         '+ %0.0f mod %0.0f has order %0.0f.'], EllipticCurve.a, ...
         EllipticCurve.b, ...
         EllipticCurve.p, EllipticCurve.Order);
fprintf(['\nBase point (%0.0f, %0.0f), generate the following subgroup:', ...
         '\n\n'], Subgroup.BasePoint(1), Subgroup.BasePoint(2))

fprintf('\t0P = 0(%0.0f, %0.0f) = 0\n',  Subgroup.BasePoint(1), ...
        Subgroup.BasePoint(2));

% While the point at infinity has not been reached, add 'currPoint' and the
% base point.
i = 1;
while ~isnan(currPoint)
    elements = [elements; currPoint];
    fprintf('\t%0.0fP = %0.0f(%0.0f, %0.0f) = (%0.0f, %0.0f)\n',  i, i, ...
            Subgroup.BasePoint(1), Subgroup.BasePoint(2), currPoint(1), ...
            currPoint(2));
    currPoint = AddPoints(currPoint, Subgroup.BasePoint, EllipticCurve);
    i = i + 1;
end
fprintf('\t%0.0fP = %0.0f(%0.0f, %0.0f) = 0\n',  i, i, ...
        Subgroup.BasePoint(1), Subgroup.BasePoint(2));

order = size(elements,1)+1;
mult = EllipticCurve.Order / order;
fprintf(['\nThis subgroup has %0.0f elements, which is a multiple of the', ...
         'order of the parent group (%0.0f/%0.0f = %0.0f)\n'], order, ...
         EllipticCurve.Order, order, mult);
% Return the order of the subgroup and all of its elements
end
```

B Appendix B: ECC Encryption (MATLAB Code)

Below is the code to generate the results from section four **Section 4. Elliptic Curve Cryptosystems: Theory and MATLAB Implementation**.

B.1 Application Driver: EncryptMain.m

```
% The following code provides a simple example implementation of elliptic
% curve cryptography (ECC) for message transmission with customizable curve
% parameters and base points.

%% PART 1: CREATE AN ELLIPTIC CURVE AND A CYCLIC SUBGROUP
% Create an elliptic curve in the form y^2 = x^3 + ax + b mod p, where
% a is 'EllipticCurve.a', b is 'EllipticCurve.b', and p is the 'primeNo'th
% prime number. The default curve is y^2 = x^3 + x + 1 mod 436.
EllipticCurve = struct;
primeNo = 90;
EllipticCurve.a = 1;
EllipticCurve.b = 1;
EllipticCurve.p = nthprime(primeNo);

% Use the point [0, 1] as the base point to generate a subgroup of the
% curve.
Subgroup.BasePoint = [0  1];

% Do a naive search and return all the elements generated by the base
% point and the order of the subgroup generated by the base point (this
% implementation for finding the subgroup order is very inefficient, see
% Schoof's algorithm for a feasible implementation for more complex curves).
[order, elements] = FindSubgroupOrder(Subgroup, EllipticCurve);
Subgroup.Order = order;
PrintEllipticCurveSummary(primeNo, Subgroup, EllipticCurve)

%% STEP 2: Generate the public and private keys for "Bob" and "Alice," who are
% trying to send messages securely.
[alicePrivateKey, alicePublicKey] = GenerateKeyPair(Subgroup, EllipticCurve);
[bobPrivateKey, bobPublicKey] = GenerateKeyPair(Subgroup, EllipticCurve);
PrintPublicPrivateKeys(alicePrivateKey, alicePublicKey, bobPrivateKey, bobPublicKey)

%% STEP 3: GENERATE THE COMMON KEY USING PUBLIC AND PRIVATE KEYS
% Generate the common key, and notice that it is the same (on complex curves,
% determining k_a * K_b * Q from k_a * Q and k_b * Q is infeasible, which is the
% basis for ECC's security).
aliceCommonKey = GenerateCommonKey(bobPublicKey, alicePrivateKey, Subgroup, ...
                                   EllipticCurve);
bobCommonKey = GenerateCommonKey(alicePublicKey, bobPrivateKey, Subgroup, ...
                                 EllipticCurve);
PrintCommonKeys(aliceCommonKey, bobCommonKey)

%% STEP 4: Encrypt a message using Alica's public key.
% Here, each letter in 'message' is converted to unicode and is encrypted using
% two points on the elliptic curve. First, a random number, 'l', from 0 to the
% order of the  subgroup is selected, and used as a scalar multiple of the base
% point Q. This is the first point in the pair generated for each letter. The
% second point  consists of 'l' multiplied by Alice's private key (k_a * Q),
% added to the  message, 'M'. Technically, 'M' should be an element of the
% subgroup, but for  the sake of simplicity I selected the point M = (unicode
% of letter, 0). This isn't quite correct, but it still gets the idea across.
% f('message') = M should be a mapping from the message text to the subgroup.
% Here, I didn't  include such a mapping because this program is designed to
% work with small  curves, where the subgroup order is less than the total
% possible unicode values.
message = 'Hello world';
Encryption = EncryptMessage(message, Subgroup, alicePublicKey, EllipticCurve);

%%
PrintEncryptionProcess(message, alicePublicKey, Encryption)

%% STEP 5: Decrypt the message using Alice's private key.
Decryption = DecryptMessage(Encryption, Subgroup, alicePrivateKey, ...
                            EllipticCurve);
```

```
65
66                          %%
67 PrintDecryptionProcess(alicePrivateKey, Decryption, Encryption)
```

B.2 Main Function 1: AddPoints.m

```
1  % Add point 'p' to point 'q' on curve 'curve.' Return the resultant
2  % point, 'r,' and whether 'p' + 'q' has reached the point at infinity by the
3  % boolean 'reachedInfinity.' If the point at infinity has been reached, 'r'
4  % will have a nan value and 'reachedInfinity' will be true.
5  function r = AddPoints(p, q, EllipticCurve)
6  x1 = p(1);
7  y1 = p(2);
8  x2 = q(1);
9  y2 = q(2);
10
11 % If the points are the same, lambda = (3*x1^2 + a) / (2 * y1).
12 if isequal(p,q)
13     inv = ModMultInverse(2 * y1, EllipticCurve.p);
14     if ~isnan(inv)
15         lambda = (3*x1^2 + EllipticCurve.a) * inv;
16     end
17 % If the points are different, lambda = (y2 - y1) / (x2 - x1).
18 else
19     inv = ModMultInverse(x2-x1, EllipticCurve.p);
20     if ~isnan(inv)
21         lambda = mod(y2 - y1, EllipticCurve.p) *  inv;
22     end
23 end
24
25 r = nan;
26
27 % If the point at infinity has not been reached, assign the correct values to
28 % 'r.'
29 if ~isnan(inv)
30     lambda = double(lambda);
31     x3 = lambda^2 - x1 - x2;
32     y3 = lambda*(x1-x3) - y1;
33     x3 = mod(x3, EllipticCurve.p);
34     y3 = mod(y3, EllipticCurve.p);
35     r = [x3 y3];
36 end
37 end
```

B.3 Main Function 2: MultiplyPoints.m

```
1  % Add the base point 'publicPoint' to a point in the subgroup, 'q', 'n' times
2  % on curve 'curve'. If the point at infinity has been reached, wrap around
3  % to the base point.
4  function p = MultiplyPoints(basePoint, p, n, EllipticCurve)
5  i = 1;
6  while i < n
7      p = AddPoints(p, basePoint, EllipticCurve);
8      i = i+1;
9      if isnan(p)
10         i = i+1;
11         p = basePoint;
12     end
13 end
14 end
```

B.4 Main Function 3: ModMultInverse.m

```
1  % Return the modular multiplicative inverse of 'x' mod 'p.' If the point at
2  % infinity has been reached, return an inverse value of nan.
3  function inverse = ModMultInverse(x,p)
4
5  % Convert x and p to higher-precision storage
6  x = sym(x);
7  p = sym(p);
8
9  % Find the modular multiplicative inverse gcd and mod.
```

```
[G, C, ~] = gcd(x,p);
if G == 1
    inverse = mod(C,p);

    % Convert the modular multipicative inverse back into double precision
    inverse = double(inverse);
else
    inverse = nan;
end
end
```

B.5 Main Function 4: FindSubgroupOrder.m

```
% Find the order and all the elements generated by base point 'q' on curve
% 'curve'.
function [order, elements] = FindSubgroupOrder(Subgroup, EllipticCurve)

% Create the 'elements' array which will hold all the elements in the
% subgroup and set the initial 'currPoint' value to the base point
elements = zeros(0,2);
currPoint = Subgroup.BasePoint;

% While the point at infinity has not been reached, add 'currPoint' and the base
% point.
while ~isnan(currPoint)
    elements = [elements; currPoint];
    currPoint = AddPoints(currPoint, Subgroup.BasePoint, EllipticCurve);
end

% Return the order of the subgroup and all of its elements
order = size(elements,1);
end
```

B.6 Main Function 5: GenerateKeyPair.m

```
% Generate a private key by multiplying the base point by a random scalar
% between 1 and the order of the group
function [privateKey, publicKey] = GenerateKeyPair(Subgroup, EllipticCurve)
privateKey = randi(Subgroup.Order);
publicKey = MultiplyPoints(Subgroup.BasePoint, Subgroup.BasePoint, ...
                           privateKey, EllipticCurve);
end
```

B.7 Main Function 6: GenerateCommonKey.m

```
% Generate the common key (k_a * k_b * Q) by multiplying the other person's
% public key (k_b * Q, in Alice's case) with the private key (k_a, in Alice's
% case). This common key, k_a * k_b * Q, will be the same for both parties
% exchanging keys.
function CommonKey = GenerateCommonKey(publicKey, privateKey, Subgroup, ...
                                       EllipticCurve)

CommonKey = MultiplyPoints(Subgroup.BasePoint, publicKey, privateKey, ...
                           EllipticCurve);
end
```

B.8 Main Function 7: EncryptMessage.m

```
% Encrypt the plain text message 'message' using public key 'publicKey,'
% generated in subgroup 'Subgroup' on curve 'EllipticCurve.' Encrypt each letter
% individually, and return a pair of points corresponding to the encrypted value
% of each characcter. Encryption.Keys contains the points corresponding to
% lQ, where l is a random number less than the subgroup order.
% Encryption.Message contains the points corresponding to M + l(kQ), where M
% is the point (unicode, 0), where 'unicode' is the unicode value of the letter,
% l is the random integer, and kQ is the publi key.
function Encryption = EncryptMessage(message, Subgroup, publicKey, ...
                                     EllipticCurve)

unicodeValues = double(message);

```

```
Encryption = struct;
Encryption.Keys = zeros(2, length(unicodeValues));
Encryption.Message = zeros(2, length(unicodeValues));
for i = 1:length(unicodeValues)
    randVal = randi(Subgroup.Order);

    currKey = MultiplyPoints(Subgroup.BasePoint, Subgroup.BasePoint, ...
                             randVal, EllipticCurve);
    Encryption.Keys(1:2,i) = currKey;

    randValEncryptionPair = MultiplyPoints(Subgroup.BasePoint, publicKey, ...
                                           randVal, EllipticCurve);
    currentCharacter = [unicodeValues(i) 0];
    encryptedCharacter = currentCharacter + randValEncryptionPair;
    Encryption.Message(1:2,i) = encryptedCharacter;
end
end
```

B.9 Main Function 8: DecryptMessage.m

```
% Decrypt the message contained in 'Encryption.' For each letter, generate
% k(lQ) by multiplying the corresponding column in Encryption.Keys (which is
% lQ) by the private key. Subtract this from the corresponding column in
% Enryption.Message to find M + l(kQ) - k(lQ) = M. Convert the x coordinate
% of M from unicode to plain text to find the unencrypted letter.
% After this process has been completed for each letter, save the final
% plaintext message in 'Decryption.Message.'
function Decryption = DecryptMessage(Encryption, Subgroup, privateKey, ...
                                     EllipticCurve)
Decryption = struct;
Decryption.FirstPoint = zeros(2, size(Encryption.Keys,2));
Decryption.UnicodeMessage = [];
for i = 1:size(Encryption.Keys,2)
    currentKey = Encryption.Keys(1:2,i);
    currentEncryptedMsg = Encryption.Message(1:2,i);

    secretRand = MultiplyPoints(Subgroup.BasePoint, currentKey, privateKey, ...
                                EllipticCurve);
    Decryption.FirstPoint(1:2,i) = secretRand;

    msg = currentEncryptedMsg - secretRand';
    Decryption.UnicodeMessage = [Decryption.UnicodeMessage msg];
end
Decryption.Message = char(Decryption.UnicodeMessage(1,:));
end
```

B.10 Helper Function 1: PrintEllipticCurveSummary.m

```
% Print the curve, base point, and the order of the subgroup.
function PrintEllipticCurveSummary(primeNo, Subgroup, EllipticCurve)
fprintf(['\n(STEP 1) Encryption using eliptic curve y^2 = x^3 + %0.0fx',...
         '+ %0.0f mod %0.0f '], EllipticCurve.a, EllipticCurve.b, ...
         EllipticCurve.p);
fprintf('\n\t... Where %0.0f is the %0.0fth prime number', EllipticCurve.p, ...
        primeNo);
fprintf(['\n\t... And base point (%0.0f, %0.0f) generates a subgroup of ', ...
         'order %0.0f\n'], Subgroup.BasePoint(1), Subgroup.BasePoint(2), ...
         Subgroup.Order)
end
```

B.11 Helper Function 2: PrintPublicPrivateKeys.m

```
% Print Alice and Bob's public and private keys
function PrintPublicPrivateKeys(alicePrivateKey, alicePublicKey, bobPrivateKey, bobPublicKey)
fprintf('\n(STEP 2) Creating Public and Private Keys...\n')
fprintf('\tAlice''s Secret Key (k_a): %3.0f\n', alicePrivateKey)
fprintf('\tAlice''s Public Key (k_a * Q): (%3.0f, %3.0f)\n', ...
        alicePublicKey(1), alicePublicKey(2))
fprintf('\tBob''s Secret Key (k_b): %3.0f\n', bobPrivateKey)
fprintf('\tBob''s Public Key (k_b * Q): (%3.0f, %3.0f)\n', bobPublicKey(1), ...
        bobPublicKey(2))
end
```

B.12 Helper Function 3: PrintCommonKeys.m

```
% Print the common keys that Alice and Bob generate using their own private key
% from each other's public key
function PrintCommonKeys(aliceCommonKey, bobCommonKey)
fprintf('\n(STEP 3) Generating Common Keys...');

fprintf('\n\tAlice''s Common Key (P = k_a * k_b * Q): (%3.0f, %3.0f)', ...
        aliceCommonKey(1), aliceCommonKey(2));
fprintf('\n\tBob''s Common Key (P = k_b * k_a * Q): (%3.0f, %3.0f)\n',...
        bobCommonKey(1), bobCommonKey(2));
end
```

B.13 Helper Function 4: PrintEncryptionProcess.m

```
% Print the encription process.
function PrintEncryptionProcess(message, alicePublicKey, Encryption)
fprintf(['\n(STEP 4) Encrypting message "%s" using Alice''s public ', ...
         'encryption key: \n\t\t k_a * Q= (%0.0f, %0.0f) '], message, ...
         alicePublicKey(1), alicePublicKey(2));
fprintf(['generates the following pairs of points \n\t\t (one set of ', ...
         'points, {(l * Q), (M + l(k_a * Q))}, per letter):\n'])
for i = 1:size(Encryption.Message,2)
    fprintf('\t%2.0f. ''%s'' (Unicode %3.0f), l=%3.0f -> ', i, ...
            message(i), double(message(i)), Encryption.RandInt(i));
    fprintf('{(%3.0f(0, 1)), ((%3.0f, 0) + %3.0f(%3.0f, %3.0f))}', ...
            Encryption.RandInt(i), double(message(i)), Encryption.RandInt(i),...
            alicePublicKey(1), alicePublicKey(2))
    fprintf('\n\t\t = {(%3.0f, %3.0f), (%3.0f, %3.0f)}\n', Encryption.Keys(1,i), ...
            Encryption.Keys(2,i), Encryption.Message(1,i), ...
            Encryption.Message(2,i));
end
end
```

B.14 Helper Function 5: PrintDecryptionProcess.m

```
% Print the decryption process and decrypted message
function PrintDecryptionProcess(alicePrivateKey, Decryption, Encryption)
fprintf(['\n(STEP 5) Decrypting message using Alice''s private key, k_a = ', ...
         '%0.0f...\n'], alicePrivateKey);
for i = 1:size(Decryption.Message,2)
    fprintf(['\t%2.0f. k_a(l * Q) \t\t\t\t\t= %3.0f(%3.0f, %3.0f) ', ...
             '\t\t\t= (%3.0f, %3.0f)'], i, alicePrivateKey, ...
             Encryption.Keys(1,i), Encryption.Keys(2,i), ...
             Decryption.FirstPoint(1,i), Decryption.FirstPoint(2,i));
    fprintf(['\n\t\t(M + l(k_a * Q)) - k(l * Q) \t= (%3.0f, %3.0f) - ', ...
             '(%3.0f, %3.0f) \t= (%3.0f, %3.0f) -> ''%s''\n'], ...
             Encryption.Message(1,i), Encryption.Message(2,i), ...
             Decryption.FirstPoint(1,i), Decryption.FirstPoint(2,i), ...
             Decryption.UnicodeMessage(1,i), Decryption.UnicodeMessage(2,i),...
             Decryption.Message(i))
end
fprintf('\n\tFully decrypted message: %s\n', Decryption.Message)
end
```