Computer Science

Title: Investigating The Security and Efficiency of RSA and Elliptic Curve Encryption Protocols

Research Question: How does the strength of the backdoor and communication efficiency of the RSA algorithm compare to those of Elliptic Curve Cryptography?

May 2023

Word Count: 3373

Candidate Code: kly215

Table of Contents

Introduction

The internet has become the ultimate medium for all types of communication and sharing: social media, text messaging, cloud-based sharing, smart homes… have revolutionized our lifestyles. And with today's digital age comes an introduction to Big Data and an increasing reliance on information technology where profit is concerned. Businesses are increasingly adopting large, computerized data banks and depend on the circulation of huge volumes of sensitive data over the web (Rivlin & Litan, 2001). Thus, concerns regarding data security are going rampant: integrity, authenticity, and confidentiality prove to be key issues for organizations to consider in the digital environment. To put these hazards at bay, organizations encrypt their sensitive data before broadcasting them around.

However, choosing the right encryption algorithm is a critical decision such businesses have to take. Two factors have to be taken into account when making these choices. Those are the strength of security they need and the budget available to allocate towards such an encryption scheme.

Cryptography has become a standard practice in all forms of online communication. This Extended Essay introduces Public-key cryptography through two well-known algorithms: RSA (Ron Rivest, Adi Shamir, Leonard Adleman) and ECC (Elliptic curve cryptography). One experiment instigates an algorithm that breaches RSA and ECC and compares their security by measuring the time taken to crack them. The other experiment measures the time and memory taken for a message to be encrypted and decrypted between RSA and ECC.

Background Information

Encryption and Decryption

Encryption describes a function that produces a distorted, unreadable version of a message to make it incomprehensible for unintended third parties. Only the addressed receiver holds the ability to reverse the encryption and access the message. The encryption and decryption are units included in what is known as a cryptosystem. A cryptosystem is a structure that contains a set of algorithms for encoding and decoding messages (Hellman & Hellman, 1976). They contain:

1. An *encryption* algorithm that converts plaintext (unencrypted data) into ciphertext (encrypted data transformed from plaintext).
2. A communication channel over which the ciphertext is sent to the receiver, which is usually unsafe.
3. A *decryption* function that reverses the encryption.
4. A *key* is a variable value required for the encryption and decryption to function (Lefton, 1991).

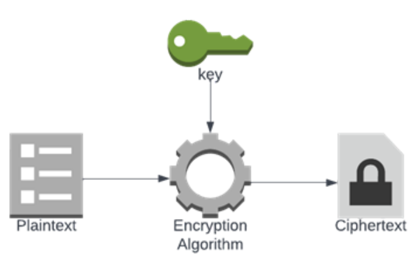


Figure 1: The Function Of An Encryption Algorithm

Encryption applies a series of mathematical operations on the plaintext to get random-looking information. Decryption applies a different mathematical operation on the random-looking information to retrieve the plaintext. There are mainly two kinds of cryptosystems depending on the nature of the key involved.

Symmetric Key Cryptosystems

The dawn of encryption was based on the sharing of one identical key between the communicating parties, hence the term "symmetric". The shared key must be communicated to initiate contact, so there becomes a risk of interception and thus sensitive data become exposed.

In addition, maintaining a large-scale symmetric encryption system requires a distinct secret key to be generated for each user. Considering that a business such as a bank must secure its communications with each of its clients, an extensive number of keys must be managed, and a lot of risky transactions must take place just to establish them (Lefton, 1991).

Nevertheless, symmetric key algorithms are still adopted, mostly where data is encrypted only for a brief, short period. Some well-known symmetric key cryptosystems are AES, DES, and RC6.

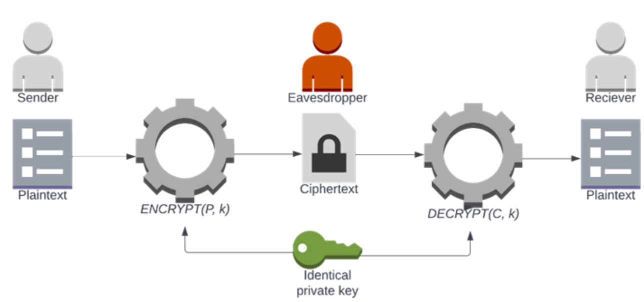


Figure 2: A Private Key Cryptosystem

Public Key Cryptosystems

Public key cryptosystems use a key pair that is mathematically linked for encryption and decryption. k_{public} can be stored anywhere: a URL, a QR code, etc. They are usually published in a public directory for everybody to access. $k_{private}$ should be encrypted and kept secret with access as limited as possible (Rivest et al., 1978). One key in the pair is published widely over the web and is stored in a digital database. The other key is not shared whatsoever and is kept only in the user's system for utmost security (Lefton, 1991).

Each user's keys are mathematically linked. The receiver's public key is used to encrypt the message intended for them. Upon decryption, the receiver uses their private key. This way, communication is done securely without a prior exchange of keys, and each user maintains only one private key instead of one for each contact.

The two Asymmetric key cryptosystems studied in this paper are the RSA algorithm and the Elliptic Curve Cryptography algorithm. Most asymmetric keys are based on the principles of the Diffie-Hellman key exchange. Therefore, an understanding of the D-H exchange is required for an understanding of RSA and ECC.

Research Question

How does the security and efficiency of the RSA and the Elliptic Curve Cryptography protocols compare in terms of time taken to encrypt, decrypt, and crack the cryptosystem?

Diffie-Hellman Key Exchange

Whitfield Diffie and Martin Hellman devised the concept of a public key cryptosystem. In 1976, they designed the Diffie-Hellman Key Exchange Protocol, where public variables are combined with some private variables to establish a shared secret (Lefton, 1991). The steps of a Diffie-Hellman key exchange include:

1. Each of the communicating parties publicly agree on two public prime numbers *n* and *g*. *n* is oftentimes large for security purposes and is often 2000 or 4000 bits long (Lefton, 1991).

2. The sender secretly generates a private key *a*; the receiver secretly generates a private key *b*

3. The sender calculates $g^{a} mod(n)$; the receiver calculates $g^{b} mod(n)$.

4. The sender and receiver exchange the keys.

5. Both parties raise the other's public key to the power of their own private key. Due to the laws of exponentiation, both the sender and receiver arrive at the same number $g^{ab} mod(n)$ (Lefton, 1991). That's their shared secret.

In most cases, The D-H key exchange is used to arrive at a private key to perform symmetric encryption. However, the idea of exchanging public variables with private ones served as the foundation of public key cryptosystems.

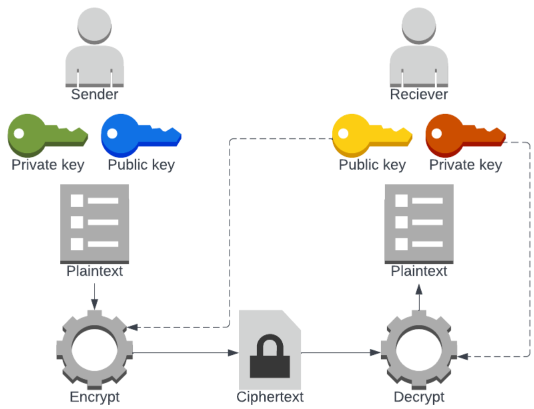


Figure 3: A Public Key Cryptosystem

RSA Encryption Algorithm

RSA introduces a practical implementation of the Diffie-Hellman key exchange using number theory (Rivest et al., 1978), and is the standard for encryption over the web as of currently (Sirajuddin, 2019, p. 4).

Rationale

Public key cryptography is based upon mathematical problems that are elaborate and unmanageable for an eavesdropper. RSA turns to modular arithmetic and prime factorization to construct an encryption function that is easily implemented but impossible to reverse without the private parameters (Rivest, 1978).

Key Generation

User A’s RSA key pair is generated by following these steps:

1. Choose two random prime numbers p and q.
    a. For utmost security, p_A and q_A must be large, at least 512 bits long.
2. Calculate the modulus $n_A = p_A * q_A$
    a. The length of n_A in bits is the key size. It is twice the bit size of p or q.
3. Calculate Euler’s totient of n, $\varphi(n)$
    a. It states that for a prime $n_A = p_A * q_A \Rightarrow \varphi(n) = (p - 1)(q - 1)$
4. Choose an integer e such that:
    a. $e < \varphi(n)$
    b. $\gcd(e, \varphi(n)) = 1$ (e and $\varphi(n)$ have no common factors)
5. Calculate an integer d such that $d \equiv e^{-1} mod(n)$, (Sirajuddin, 2019, p. 5).

Note that the modulus n and e have to be large in order to accommodate for the necessary complexity of the modular logarithm, otherwise reversing the encryption involves calculating a regular logarithm (Eckstein, 1996). This process generates a pair of asymmetric keys k, where:

$$k_{public} = n, e \qquad\qquad k_{private} = d$$

Encryption

First, the sender A has to fetch the recipient B’s public key $k_{public}(n_B, e_B)$ in a public directory. It can be thought of as an “open lock”. Then, The sender converts P into a numerical equivalent p, using an agreed upon conversion method such as Unicode or UTF-8 (Lefton, 1991). Finally, the sender computes the ciphertext C from p and $k_{public}(n, e)$.

$$C \equiv P^{e_B} mod(n)$$

In doing so, the sender closes the "lock" with their message and sends it to the receiver.

Decryption

The trapdoor is a variable that makes it simple to reverse a one-way function. The rightful recipient of the message has this trapdoor. In RSA, it is $k_{private}(d_B)$ (Eckstein, 1996).

The recipient receives the ciphertext $C \equiv p^{e_B} mod(n)$, but they need to decrypt the message to be able to make sense of it. So, they need another exponent d_B that will undo the encryption. Knowing that d and e are inverses[1], the numerical equivalent of the plaintext P is obtained.

$$C^{d_B} \equiv {P^{e_B}}^{d_B} mod(n) \Rightarrow P^{e_B d_B} mod(n) = P$$

Prime Factorization And Backdoors

A backdoor is a function or algorithm that intends to calculate the trapdoor from public variables, compromising security in doing so (QuintessenceLabs, 2022).

Factoring primes is a daunting tasks, especially when said primes are large numbers. The current standard for RSA is 2048 bit security, which describes a modulus n that would take roughly 300 trillion years to factor, which is about 22000 times the age of the universe.

[1] Understanding how d is generated requires an understanding of the Extended Euclidean theorem and Bézout's identity, see Rivest et al., 1977, pp. 6-8

Despite the public belief that the public modulus n_A is a prime number, it is not. The definition of a prime number states that it has no factors other than one and itself. n_A has factors other than 1 and itself, that being p_A and q_A. Instead, it is close to a prime number, or pseudorandom, and this opens opportunities for backdoors.

Mathematicians have devised techniques that aid in the factoring process, such as the Chinese Remainder Theorem, and other people have already devised backdoors for RSA. See Crépeau, Slakmon, (2003). Simple Backdoors for RSA Key Generation. Lecture Notes in Computer Science.

Elliptic Curve Encryption Algorithm

Elliptic curve cryptography is a public-key cryptosystem that bases itself on the rules and structure of elliptic curves. An elliptic curve *(E)* is a projective algebraic curve that satisfies the following relation known as the Weierstrass equation (Chatzigiannakis et al., n.d.):

$$y^2 + axy + by = x^3 cx^2 + dx + e \ (mod\ p)$$

For the sake of simplicity in this Extended Essay, some constants will be set to be zero or one. A frequently used equation that defines an elliptic curve is as follows:

$$y^2 = x^3 + ax + b \ (mod\ p)$$

Elliptic curves look as follows:

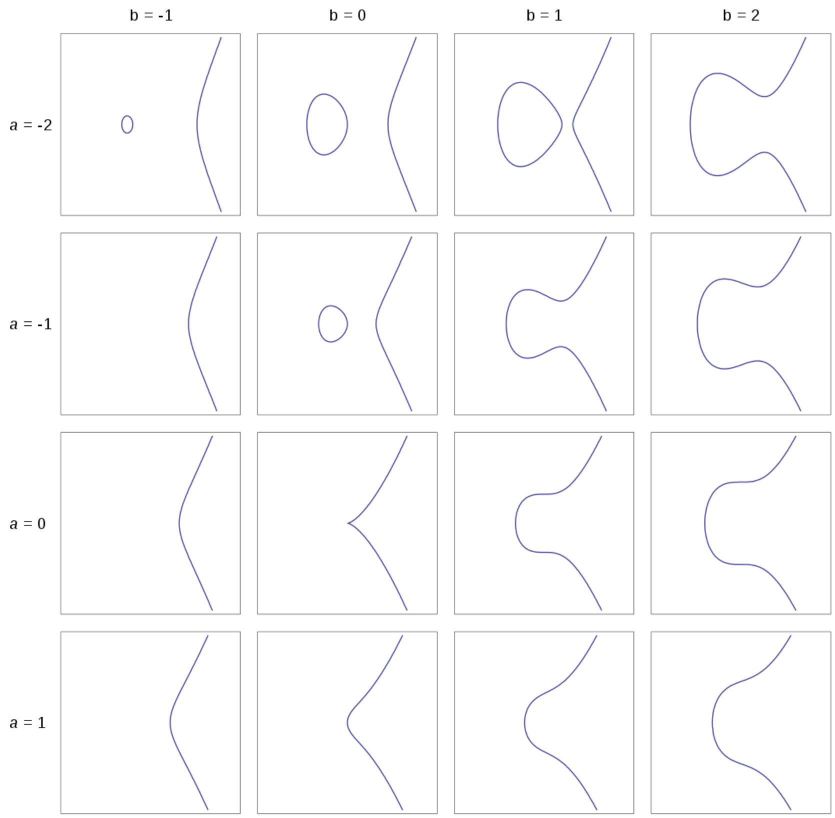


Figure 4: Elliptic Curve Catalog (*Elliptic curve catalog* 2008)

Choosing elliptic curve constants isn't as easy compared to Diffie-Hellman and RSA. Randomly generating *a* and *b* will yield a weaker curve compared to another elliptic curve with well-selected constants.

Rationale

The basic notion of elliptic curve encryption is carrying out a series of multiplications with points on the curve. Elliptic curves allow carrying out such arithmetic operations on points easily. For an eavesdropper, doing the reverse is impractically difficult (Cromwell, 2022).

To be properly implemented, elliptic curves generate keypairs over a set of numbers referred to as domain parameters. They are as follows:

1. *q* the size of the field over which the curve is generated (for this Extended Essay, the set of finite numbers)
2. *a* and *b*, elliptic curve parameters
3. A generator / base point $G(x_G, y_G)$
4. The order *n* of the base point *G*, to be explained in the Encryption section.

It is recommended to periodically alter the domain parameters for amplified security (Adalier & Teknik, 2015).

Point Multiplication

Elliptic Curve Cryptography is based upon the operation of point addition function, where point *A* added to a point B on the curve and yields a third point $-C(x_c, -y_c)$ on the curve, which is reflected vertically on the symmetrical image of the curve to obtain point $C(x_c, y_c)$ (Gura, Patel, Wander, Hans, & Shantz).

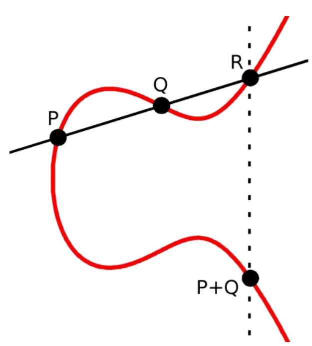


Figure 5: Point Addition On Elliptic Curves (Sutherland, 2017)

When points A and B are coinciding points (have the same coordinates; are the same point), this operation becomes known as point multiplication, or “dotting”. It is performed in a similar fashion to point addition, but instead of adding two points, adding a point to itself yields the tangent to the curve at that point. The other intersection of said tangent with the curve gives the point $-\ C$ (Wagner, 2020).

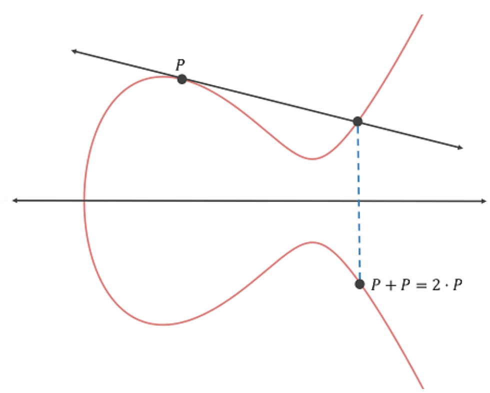


Figure 6: Point Multiplication On Elliptic Curves (Yang, 2022)

Key Generation

Public Variables

1. Elliptic curve parameters *a* and *b*.
2. Choose a generator point $G(x_{G}, Y_{G})$ on *(E)*.

Process

1. User A selects a private key n_A in the field of the curve; n_A has to be a large prime number.
2. They calculate their public key $P_A = n_A G$ based on the principles of point multiplication.

$$k_{public} = P_A(x_A, y_A) \quad k_{private} = n_A$$

Encryption

First, Like RSA, The characters of the plaintext message *p* are translated to numbers using an agreed-upon character to value conversion method such as Unicode or UTF-8.

This number is encoded as a point P_M on *(E)*. The sender calculates kG and kP_B where k is a large, random prime number that A decides and P_B is the receiver B's public point.

The points representing the ciphertext are (K, 2020):

$$C_M = \{kG,\ P_M + kP_B\}$$

Decryption

The intended receiver gets two points: $C_M = \{kG,\ P_M + kP_B\}$. They multiply the first point with their private key n_B to obtain $kG(n_B)$.

Then, they subtract the second point from their answer:

$$= P_M + kP_B - kn_B G$$

$$(\text{But } P_M = n_B G)$$

$$= P_M + k(n_B G) - kn_B G$$

$$= P_M$$

And like so, the receiver arrives at the plaintext message P_M (K, 2020).

The Discrete Logarithm Problem

G and *Q* are not integers, so *n* can not be found by dividing *G* by *Q*. Instead, an eavesdropper would have to find the multiplicative inverse.

Even if an eavesdropper keeps multiplying *G* by itself to get to *Q*, they will run into many false values due to the symmetric quality of elliptic curves. ECC deals with huge numbers, meaning that the eavesdropper will run into innumerable false solutions. And like so, cracking becomes a nearly impossible task. (Cromwell, 2022).

Experiment

The aim of this experiment is to quantify the level of security and degree of efficiency for both RSA and ECC for comparison.

The security of an algorithm depends on its key size and the design of the cryptosystem. A bigger key size increases the number of keys an attacker has to generate and try. Then, more steps and time is required to successfully crack the encryption. Even though two different algorithms are set to use the same key size, one can provide better security than the other, depending on the design of the system itself.

The amount of memory used and the time allocated for a trial encryption and encryption serves as an indicator of an algorithm's security. Speed also depends on key size, but also on the mathematical functions that underpin the algorithm. The more computationally intensive, the more time and memory it will require, and thus, less efficient.

ECC provides similar security levels to traditional algorithms such as RSA but with smaller key sizes, and therefore is optimal for devices that have limited computational power and miniscule wireless communication protocols (Chatzigiannakis et al.). In addition, the invulnerability of ECC compared to RSA has been established.

Hypothesis

Elliptic Curve Cryptography provides better security and is more efficient than RSA in terms of speed and memory.

Methodology

To test security, A key pair will be generated in Java. Next, the program will start to calculate the private variables from the public key. Once it successfully cracks the algorithm, it measures the duration and memory it took to do so. The harder it is to crack the algorithm, or in other words, the greater the duration, the better the algorithm.

To test efficiency, a cryptosystem will be set up in Java over which a plaintext message will be encrypted and then decrypted. Measuring the duration and memory it takes to do this task will provide insight into the effectiveness of the encryption algorithm tested. The less time and memory the program demands, the better the algorithm.

Java is the programming language chosen for this paper, and the IDE is IntelliJ IDEA. This experiment is done on a machine with an 11th Generation Intel i5 processor, 8 GB of RAM, 512 GB of SSD storage, and a Windows 11 Home Operating System.

The System class in Java provides a method for measuring time in milliseconds called System.currentTimeMillis(). A method for calculating the memory allocated for a program is adopted from an external source (Lars Vogel, 2008).The codes for RSA and ECC are adopted from external sources, but the cracking algorithms have been adopted or self-generated.

Security Test

RSA

For this experiment, a Java program "RsaKeyGenerator.java" was adopted (Yang, 2013, par 4). The original code only generates the key pair. A factoring algorithm has been added (OldCurmudgeon, 2013) as well as functionality that measures the elapsed time and memory allocated for brute forcing.

Pseudocode

"

```
 PHIn, E, D

   method main
       RND = new Random
       SC = new Scanner(System.in)
       Output prompt "Enter the bit size of p and q"
       // Individual bit sizes of p and q.
       // The bit size of n = p*q is double this bit size since
it is the product.
       input BITSIZE
```

```
        // Generates a prime number with the bit size specified
by the first parameter
        P, Q = random prime of length BITSIZE using RND
        // Yields the public modulus n.
        N = P*Q
        OUTPUT N, N.bitLength

        // What follows in the main method is generated by the
student for the Extended Essay.
        // Start of factoring the public modulus n
        // Measures the starting time
        STARTTIME = current system time

        factors(N, false)
        // After successfully factoring n, the security is
compromised.
        // Measures the end time
        ENDTIME = current system time
        // The difference between the end time and the starting
time is the duration of the factoring.
        ELAPSEDTIME = ENDTIME - STARTTIME
        Output ELAPSEDTIME

        // Measures the allocated memory
        // Get the Java runtime
        RUNTIME = Runtime.getRuntime()
        // Run the garbage collector
        RUNTIME.gc()
        // Calculate the used memory
        MEMORY = RUNTIME.totalMemory() -RUNTIME.freeMemory()
        Output "Used memory (Bytes): " + memory

    end main

    method factors(N, DUPLICATES)
        // Have we done this one before?
```

```
        F = new Collection()
        if F is empty
            // Start empty.
            // Check for duplicates.
            LAST = 0
            // Limit the range as far as possible.
            loop from I = 2 till N/I
                // Can have multiple copies of the same factor.
                while n.mod(I) equals 0
                    if DUPLICATES Or I ≠ LAST
                        F.add(I)
                        LAST = I
                    end if
                    // Remove that factor.
                    n /= I
                end while
            end loop
            if n > 1
                // Could be a residue.
                if DUPLICATES Or N ≠ LAST
                    f.add(n)
                end if
            end if
        end if
        Output "Prime factors of the modulus n: "+F.toString()
    end factors
end class
```
”

Notes

- The program asks the user to input their desired bit size for *p* and *q*
    - Since $n = p * q$, *n* has a bit size that is twice the user-generated value.
- factors() is a method that takes the public variable *n* and factors it.

- The program calculates the duration it took to factor *n* and thus crack the encryption.

Elliptic Curve Encryption Algorithm

For this experiment, a Java program "crackECC.java" was adopted (Azaky, 2015). designed with the following pseudocode. The original code only generates the key pair. The cracking algorithm was developed by the student.

Pseudocode

"

```
class main
    AUXILIARY_CONSTANT_LONG = 1000
    AUXILIARY_CONSTANT =
BigInteger.valueOf(AUXILIARY_CONSTANT_LONG)

    // The return type is changed from KeyPair to long to return
the cracking duration.
    Method generate_and_crack_KeyPair(EllipticCurve C, Random
RND, Scanner SC)
        // Randomly select the private key, such that it is
relatively prime to P
        P = c.getP()
        PRIVATEKEY

        // Gather the key bit size (student-generated)
        Output "Enter the bit size of the private key"
        PRIVATEKEY = new BigInteger(input BITSIZE, RND)

        // Calculate the public key, k * g.
        // First, randomly generate g if it is not present in the
curve.
        G = C.getBasePoint()
        if G is null
            // Randomly generate g using Koblits method.
```

```
            // The starting value of x should be random.
            X = new BigInteger(P.bitLength(), RND)
            G = koblitzProbabilistic(C, X)
            C.setBasePoint(G)
        End if
        PUBLICKEY = C.multiply(G, PRIVATEKEY)

        // What follows in this method is generated by the
student for the Extended Essay
        // Bruteforce the keypair by trying to find the public
point from the generator
        // Measures the starting time
        STARTTIME = current system time

        // Setup the cracking algorithm by marking the generator
point
        I = 1
        TR = C.multiply(G,I)
        I++

        // Loop that keeps adding G to itself till the public
point is reached
        // Counts the number of additions through i
(student-generated).
        While TR.X.intValue() ≠ PUBLICKEY.x.intValue() &&
TR.Y.intValue() ≠ PUBLICKEY.Y.intValue())
            TR = C.multiply(G, I)
            I++
        End while

        // Measures the end time (student-generated).
        ENDTIME = current system time

        // Calculates the difference, the elapsed time in
milliseconds
        ELAPSEDTIME = ENDTIME - STARTTIME
        return ELAPSEDTIME
```

```
        // Measures the allocated memory
        // Get the Java runtime
        RUNTIME = Runtime.getRuntime()
        // Run the garbage collector
        RUNTIME.gc()
        // Calculate the used memory
        MEMORY = RUNTIME.totalMemory() -RUNTIME.freeMemory()
        Output "Used memory (Bytes): " + memory
    End main

    Method koblitzProbabilistic(EllipticCurve C, BigInteger X)
        P = C.getP()


        if not P.testBit(0) OR not p.testBit(1)
            // throw an exception if P != 3 (mod 4)
        End if

        PMINUSONEPERTWO = (P - 1).shiftRight(1)

        TEMPX= X * AUXILIARY_CONSTANT modulus P
        Loop from K = 0 till AUXILIARY_CONSTANT_LONG - 1
            NEWX = TEMPX.add(K)

            // Calculates the rhs of the elliptic curve
equation, call it a
            A = C.calculateRhs(NEWX)


            // Determine whether this value is a quadratic
residue modulo p
            // It is if and only if a ^ ((p - 1) / 2) = 1 (mod
p)
            if a modulus PMINUSONEPERTWO exponent P = 1
                // We found it! Now, the solution is y = a ^ ((p
+ 1) / 4)
                Y = (A modulus P + 1).shiftRight(2)exponent p
```

```
                return new ECPoint(newX modulus P, Y
            End if
        End loop

        // If we reach this point, then no point are found
within the limit.
        // throw an exception for that no point was found within
the auxiliary constant
    End method

    Method main
        // using NIST_P_192 to test
        C = EllipticCurve.NIST_P_192
        RND = new Random()
        SC = new Scanner(System.in)

        Output "Time elapsed (ms):
"+generate_and_crack_KeyPair(C, RND, SC)

    End method
End class
```
”

Notes

- Upon startup, the program:
    - Generates a standard elliptic curve, the NIST P-192 (Standardized by the National Institute of Standards and Technology)
    - Gets the generator point
- It gathers from the user the desired key size, and generates the key pair.

- ECC is brute forced by adding G to itself until the public point P is reached. In doing so, the attacker uncovers the private key which represents the number of point additions on the curve.
- The program calculates the duration it took to find n by point multiplication and thus crack the encryption.

Efficiency Test

RSA

For this test, a Java program "RSAEncryptionDecryption.java" is adopted (Rai, 2018). This program generates an RSA keypair, plaintext message, encrypts it, then decrypts it. The original code uses a pre-defined bit size for the key. It has been changed to allow for the user to use their desired key size, measure the duration, and the memory allocated for those tasks.

Pseudocode

"

```
PUBLIC_KEY_FILE = "Public.key"
PRIVATE_KEY_FILE = "Private.key"

Method main

   STARTTIME = current system time

   KEYPAIRGENERATOR = KeyPairGenerator.getInstance("RSA")
   SC = new Scanner(System.in)
   Input BITSIZE
   keyPairGenerator.initialize(BITSIZE) //1024 used for normal
securities
       KEYPAIR = keyPairGenerator.generateKeyPair()
       PUBLICKEY = keyPair.getPublic()
```

```
        PRIVATEKEY = keyPair.getPrivate()
        Output "Public Key - " + PUBLICKEY
        Output "Private Key - " + PRIVATEKEY

        //Pulling out parameters which makes up Key
        KEYFACTORY = KeyFactory.getInstance("RSA")
        RSAPUBLICKEYSPEC = KEYFACTORY.getKeySpec(PUBLICKEY,
RSAPUBLICKEYSPEC.class)
        RSAPRIVATEKEYSPEC = KEYFACTORY.getKeySpec(PRIVATEKEY,
RSAPRIVATEKEYSPEC.class)
        Output "PubKey Modulus : " +
RSAPUBLICKEYSPEC.getModulus()
        Output "PubKey Exponent : " +
RSAPUBLICKEYSPEC.getPublicExponent()
        Output "PrivKey Modulus : " +
RSAPRIVATEKEYSPEC.getModulus())
        Output "PrivKey Exponent : " +
RSAPRIVATEKEYSPEC.getPrivateExponent())

        //Share public key with other so they can encrypt data
and decrypt those using private key(Don't share with Other)
        RSAOBJ = new RSAEncryptionDecryption()
        RSAOBJ.saveKeys(PUBLIC_KEY_FILE,
RSAPUBLICKEYSPEC.getModulus(),
RSAPUBLICKEYSPEC.getPublicExponent())
        RSAOBJ.saveKeys(PRIVATE_KEY_FILE,
RSAPRIVATEKEYSPEC.getModulus(),
RSAPRIVATEKEYSPEC.getPrivateExponent())

        //Encrypt Data using Public Key
        Array ENCRYPTEDDATA = RSAOBJ.encryptData("Anuj Patel -
Classified Information !")

        //Decrypt Data using Private Key
        RSAOBJ.decryptData(ENCRYPTEDDATA)

        ENDTIME = current system time
        ELAPSEDTIME = ENDTIME - STARTTIME
```

```
    Output "Time Elapsed (Milliseconds)"+elapsedTime)

    // Measures the allocated memory
    // Get the Java runtime
    RUNTIME = Runtime.getRuntime()
    // Run the garbage collector
    RUNTIME.gc()
    // Calculate the used memory
    MEMORY = RUNTIME.totalMemory() -RUNTIME.freeMemory()
    Output "Used memory (Bytes): " + memory
 End method

 method saveKeys(FILENAME,MOD,EXP)
     FOS = null
     OOS = null

        Output "Generating "+fileName + "..."
        FOS = new FileOutputStream(FILENAME)
        OOS = new ObjectOutputStream(new
BufferedOutputStream(FOS))

         OOS.writeObject(MOD)
         OOS.writeObject(EXP)

         Output fileName + " generated successfully"

     finally
         if(OOS ≠ null)
             OOS.close()

             if(FOS ≠ null)
                 FOS.close()
             End if
         End if
     End finally
 End method
```

```
method encryptData(DATA)

    Output "Data Before Encryption :" + data
    Array DATATOENCRYPT = DATA.getBytes()
    Array encryptedData = null

    PUBLICKEY = readPublicKeyFromFile(PUBLIC_KEY_FILE)
    CIPHER = CIPHER.getInstance("RSA")
    CIPHER.init(CIPHER.ENCRYPT_MODE, PUBLICKEY)
    ENCRYPTEDDATA = CIPHER.doFinal(DATATOENCRYPT)
    Output "Encryted Data: " + encryptedData

    return encryptedData;
End method

method decryptData(Array DATA)

    Array DECRYPTEDDATA = null

    PRIVATEKEY = readPrivateKeyFromFile(PRIVATE_KEY_FILE);
    CIPHER = CIPHER.getInstance("RSA")
    CIPHER.init(CIPHER.DECRYPT_MODE, PRIVATEKEY)
    DECRYPTEDDATA = CIPHER.doFinal(DATA)
    Output "Decrypted Data: " + DECRYPTEDDATA
End method

method readPublicKeyFromFile(FILENAME)
    FIS = new FileInputStream(new File(FILENAME))
    OIS = new ObjectInputStream(FIS)

    MODULUS = OIS.readObject()
    EXPONENT = OIS.readObject()

    //Get Public Key
```

```
        RSAPUBLICKEYSPEC = new RSAPublicKeySpec(MODULUS,
EXPONENT)
        KEYFACTORY = KeyFactory.getInstance("RSA")
        PUBLICKEY =KEYFACTORY.generatePublic(RSAPUBLICKEYSPEC)

        return PUBLICKEY

        finally
            If OIS ≠ null
                OIS.close()
                If FIS ≠ null
                    FIS.close()
                End if
            End if
        End finally
        return null
    End method

    method readPrivateKeyFromFile(FILENAME)
        FILEINPUTSTREAM = new FileInputStream(new
File(FILENAME))
        OBJECTINPUTSTREAM = new ObjectInputStream(FIS)

        MODULUS = OIS.readObject()
        EXPONENT = OIS.readObject()

        //Get Private Key
        RSAPRIVATEKEYSPEC = new RSAPrivateKeySpec(MODULUS,
EXPONENT)
         KEYFACTORY = KEYFACTORY.getInstance("RSA")
         PRIVATEKEY =
KEYFACTORY.generatePrivate(RSAPRIVATEKEYSPEC)

            return PRIVATEKEY

         finally
            If OIS ≠ null
```

```
                OIS.close()
                If FIS ≠ null
                    FIS.close()
                End if
            End if
        End finally
        return null
End method
```
”

Elliptic Curve Encryption Algorithm

For this test, a Java program "ECC.java" is adopted (Benaich, 2015). This program generates an ECC keypair, plaintext message, encrypts it, then decrypts it. The original code uses a pre-defined bit size for the key. It has been changed to allow for the user to use their desired key size, measure the duration, and the memory allocated for those tasks.

Pseudocode

"

```
PAD = 5
    R = new Random()

    POINTTABLE = List HashMap<Point, Integer>
    CHARTABLE = HashMap<Integer, Point>

    MENCODER
    MDECODER

    Method ECC(C)
        initCodeTable(C)
        this.mEncoder = new Encoder(CHARTABLE)
        this.mDecoder = new Decoder(POINTTABLE)
    End method
```

```
    method getRandom()
        return r
    End method

    method encrypt(MSG, KEY)
        C = key.getCurve()
        G = C.getBasePoint()
        PUBLICKEY =KEY.getKey()
        P = C.getP()
        NUMBITS = P.bitLength()

        K = new BigInteger(NUMBITS, getRandom())
        while k.mod(p).compareTo(BigInteger.ZERO) == 0
        SHAREDSECRET = C.multiply(PUBLICKEY, K)

        KEYHINT= C.multiply(G, K) // key to send

        Output c
        Output "Message to encrypt, m = " + MSG
        Output "Bob's public key, Pb = " + PUBLICKEY
        Output "Alice's private key, k = " + K
        Output "The encryption key, SHAREDSECRET = K * PB = " +
SHAREDSECRET)
        Output "The hint to compute sharedSecret for bob,
KEYHINT = " + KEYHINT);

        MMATRIX = MENCODER.encode(MSG)
        MMATRIX.performAddition(Helpers.toBinary(SHAREDSECRET))
        Output "sharedSecret binary format :"
        Helpers.print Helpers.toBinary(SHAREDSECRET)
        Output "4) encrypt the matrix with sharedSecret (code
addition)"
        Output MMATRIX
        return MMATRIX.toArray(Helpers.toBinary(KEYHINT))
    EMD
```

```
    Method decrypt(Array CIPHERTEXT, KEY)
        c = KEY.getCurve()
        G = C.getBasePoint()
        PRIVATEKEY = KEY.getKey();
        P = C.getP();

        KEYHINT = Point.make(CIPHERTEXT)
        SHAREDSECRET = C.multiply(KEYHINT, PRIVATEKEY)

        //get the decypted matrix
        MMATRIX = Matrix.make(CIPHERTEXT)
        Output "C = "
        Output MMATRIX
        //substract the key form the matrix

MMATRIX.performSubstraction(Helpers.toBinary(SHAREDSECRET))
        Output MMATRIX
        //decode the matrix
        Return MDECODER.decode(MMATRIX)
    End method

    method generateKeyPair(C)
        // Randomly select the private key, such that it is
relatively prime to p
        P = C.getP()
        PRIVATEKEY
        Output "Gather key size"
        SC = new Scanner(System.in)
        Input KEYSIZE
        PRIVATEKEY = new BigInteger(KEYSIZE, getRandom())

        // Calculate the public key, k * g.
        G = C.getBasePoint()
        PUBLICKEY = C.multiply(G, PRIVATEKEY);
```

```
        return new KeyPair
                new PublicKey(C, PUBLICKEY),
                new PrivateKey(C, PRIVATEKEY)

    End method

    method initCodeTable(CURVE
        CHARTABLE = new HashMap<>()
        POINTTABLE = new HashMap<>()
        Point P = CURVE.getBasePoint()
        Loop from I = 1 till 26
            P = CURVE.multiply(CURVE.getBasePoint(), I)
        End loop
        //special characters
        CHARTABLE.put(32, Point.getInfinity()) //space
        Array codeAscii = new int[]{10, 13, 39, 40, 41, 44, 46,
58, 59};
        Loop int i : codeAscii
            P = CURVE.add(P, CURVE.getBasePoint())
            CHARTABLE.put(I, P)
        End loop

        //populate the points symbol table
        Loop (KEY : CHARTABLE.keySet())
            POINTTABLE.put(CHARTABLE.get(KEY), KEY)
        End loop
    End method

    method displayCodeTable()
        charTable.forEach((cle, val) -> {
           Output CLE.intValue() + " -> " + val)
    End method

    Method main
        STARTTIME = System.currentTimeMillis()
        C = new EllipticCurve(4, 20, 29, new Point(1, 5));
```

```
        ECC = new ECC(C)
        ECC.displayCodeTable()
        MSG = "i understood the importance in principle of
public key cryptography, but it is all moved much faster than i
expected i did not expect it to be a mainstay of advanced
communications technology";
        // generate pair of keys
        KEYS = generateKeyPair(C)
        // encrypt the msg
        Array cipherText = ECC.encrypt(MSG, KEYS.getPublicKey()
        Helpers.print(cipherText)

        // decrypt the result
        PLAINTEXT = ECC.decrypt(CIPHERTEXT,KEYS.getPrivateKey()

        Output PLAINTEXT

        // What follows is generated by the student for the
Extended Essay.
       // Start of factoring the public modulus n
       // Measures the starting time
       STARTTIME = current system time
       factors(N, false)
       // After successfully factoring n, the security is
compromised.
       // Measures the end time
       ENDTIME = current system time
       // The difference between the end time and the starting
time is the duration of the factoring.
       ELAPSEDTIME = ENDTIME - STARTTIME
       Output ELAPSEDTIME
       // Measures the allocated memory
       // Get the Java runtime
       RUNTIME = Runtime.getRuntime()
       // Run the garbage collector
       RUNTIME.gc()
       // Calculate the used memory
```

```
        MEMORY = RUNTIME.totalMemory() -RUNTIME.freeMemory()
        Output "Used memory (Bytes): " + memory
End method
```
”

Results

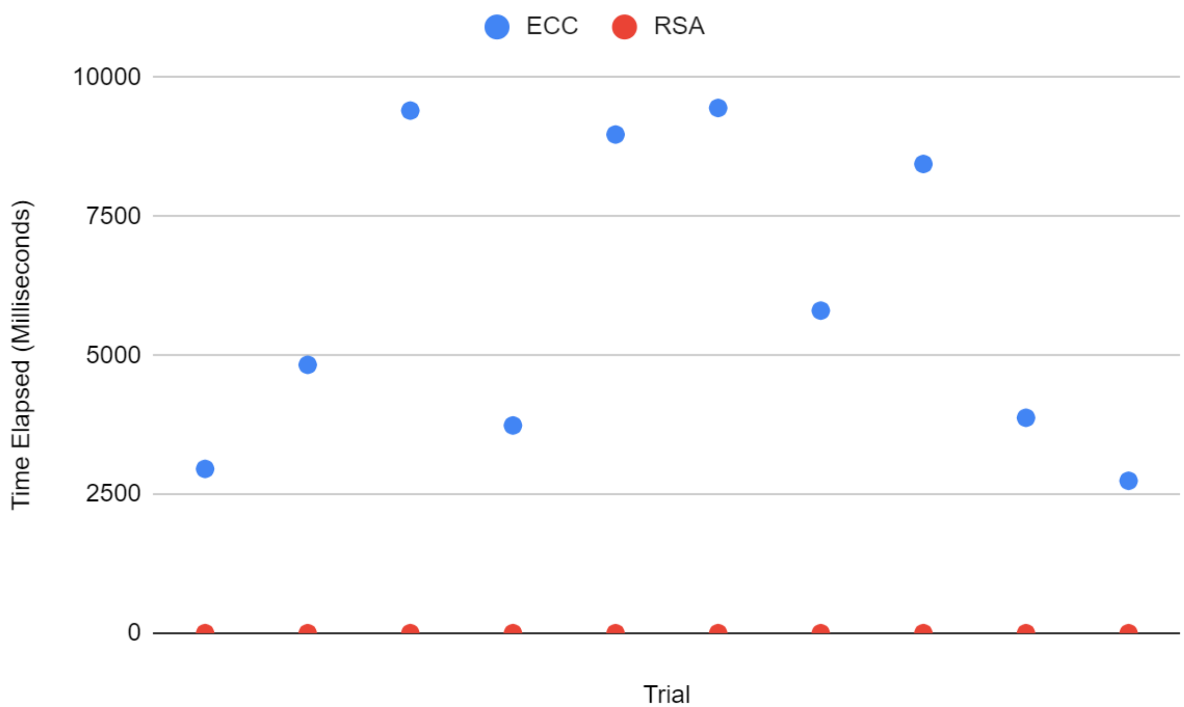
Each test was run ten times to produce the following results

Security Test

For a 16 bit key, an RSA and ECC cryptosystem was set up and put through ten trials of a brute force attack. The durations it took to breach the security of each cryptosystem is modeled in Table 1:

	Time elapsed (ms)	
Key Size (bits)	**RSA**	**ECC**
16	1	2957
	0	4830
	1	9406
	1	3739
	0	8975
	1	9453
	0	5806
	1	8445
	0	3877
	0	2743

Table 1: Time Elapsed For A Successful Brute Force Of A 16 Bit Key In RSA and ECC in Milliseconds



Graph 1: Time Elapsed For A Successful Brute Force Of A 16 Bit Key In RSA and ECC

Efficiency Test

	Time Elapsed (Milliseconds)	
Key Size (Bits)	**RSA**	**ECC**
1024	1877	3168
	2164	2535
	2524	1566
	1891	3396

	Memory Allocated (Bytes)	
Key Size (Bits)	**RSA**	**ECC**
1024	2669440	2083024
	2667144	2084272
	2597504	2083904
	2586936	2084344

	2405	1859
	1668	1551
	2591	1535
	1148	1511
	2435	1711
	1932	2748
3072	2727	2359
	2319	4730
	6727	1901
	2418	2295
	1857	1602
	2987	2466
	2241	1690
	2279	1550
	2307	1590
	2635	2787
15360	151902	4344
	37780	2696
	102624	4678
	596705	2984
	336491	2324
	191826	2636
	281475	3600
	224516	2341
	184402	2212
	65163	2226

	2588184	2083016
	2665912	2084352
	2666192	2084408
	2667104	2084184
	2594760	2084136
	2676280	2083776
3072	2635744	2119488
	2746048	2084376
	2623976	2116328
	2676008	2135728
	2636704	2088208
	2612688	2092496
	2631080	2084160
	2643848	2119552
	2593584	2120928
	2621296	2105568
15360	2662680	2119552
	2618392	2121592
	2733808	2076232
	2733976	2080008
	2695768	2119488
	2656152	2119488
	2728104	2119488
	2734976	2076096
	2706912	2119768
	2732968	2119488

Table 2: Time Elapsed (Milliseconds) For Encrypting And Decrypting A Message

Table 3: Memory Allocated (Bytes) For Encrypting And Decrypting A Message

Analysis

Security

To find a ratio between the time elapsed for RSA and ECC, the average of the ten trials has to be calculated as follows:

$$\overline{T}_{RSA} = \frac{1+0+1+1+0+1+0+1+0+0}{10} = 0.5\text{s}$$

$$\overline{T}_{ECC} = \frac{2957+4830+9406+3739+8975+9453+5806+8445+3877+2743}{10} = 6023.1\text{s}$$

	RSA	**ECC**
Average Time (Milliseconds)	0.5	6023

Table 4: Average Values Of Table 1

Based on Table 4, It takes less than 1 Millisecond to breach 16 bit key RSA security, while 16 bit key ECC stood for 6023 ms, making it the stronger contender. The ratio of the average brute forcing duration is $\frac{T_{ECC}}{T_{RSA}} = \frac{6023}{0.5} = 12,046$. This means that ECC needs 12,000 times more time to break than RSA for a 16 bit key. Therefore, it is more computationally intensive to break Elliptic Curve Cryptography than RSA.

Efficiency

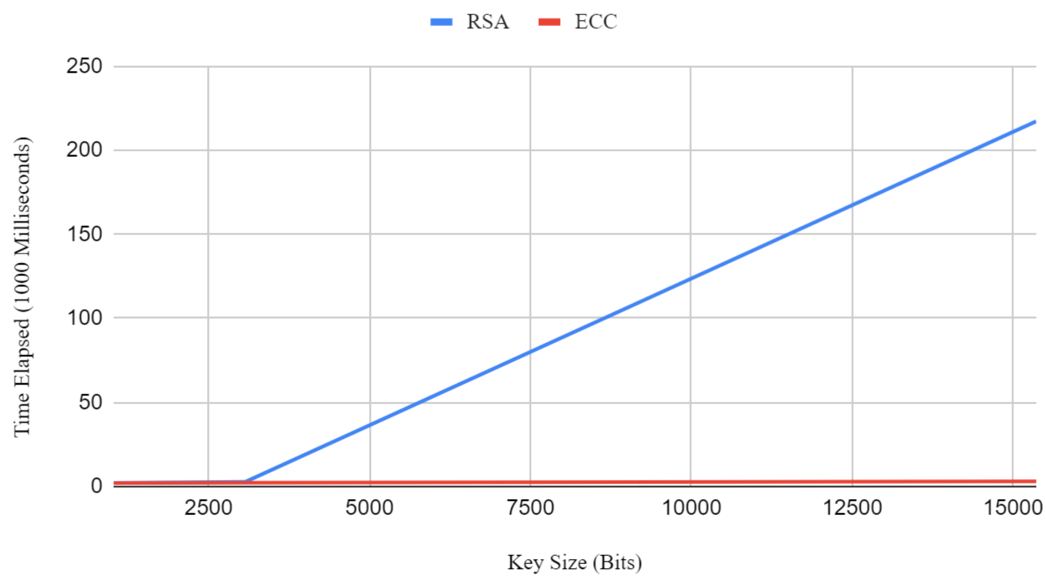
Calculating the average of the values in Table 3 and 4 gives:

Key Size (bits)	Average Time elapsed (Milliseconds) RSA	ECC
1024	2063.5	2158
3072	2849.7	2297
15360	217288.4	3004.1

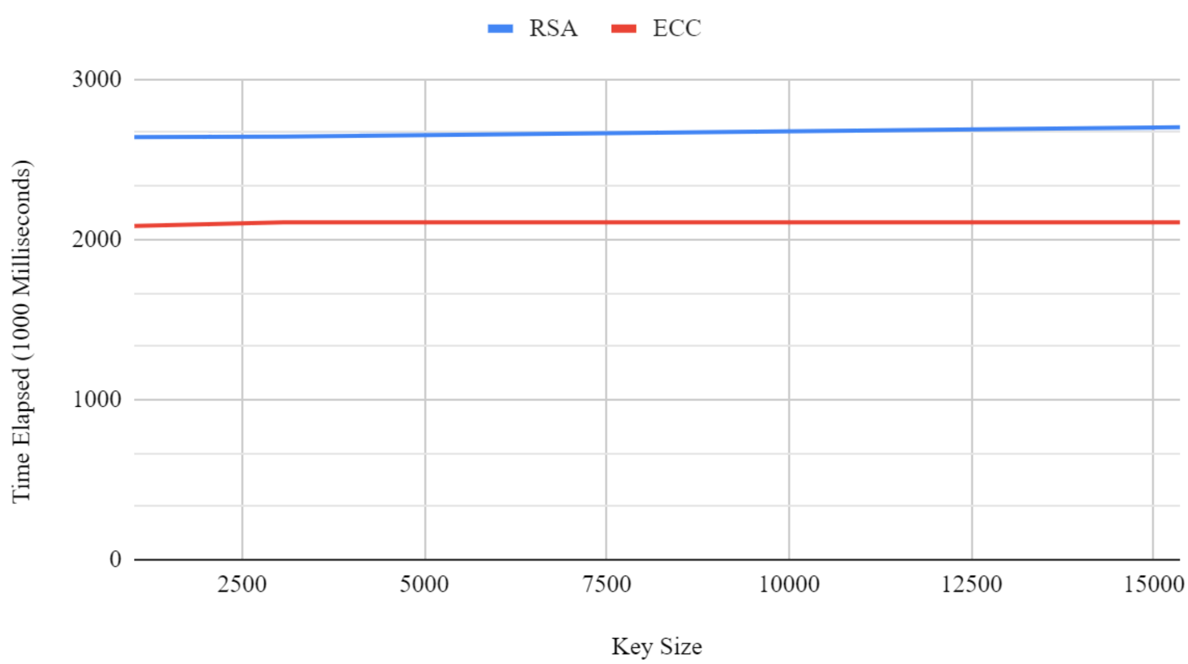
Key Size (bits)	Average Memory Allocated (Bytes) RSA	ECC
1024	2637945.6	2083941.6
3072	2642097.6	2106683.2
15360	2700373.6	2107120

Table 5: Average Time Elapsed (Milliseconds) For Encrypting And Decrypting A Message

Table 6: Average Memory Allocated (Bytes) For Encrypting And Decrypting A Message



Graph 2: Average Time Elapsed (Milliseconds) For Encrypting And Decrypting A Message in RSA and ECC



Graph 3: Average Memory Allocated (Bytes) For Encrypting and Decrypting A Message in RSA and ECC

According to Graph 2, RSA and ECC take a very similar duration to encrypt and decrypt a plaintext message for a 1024 bit key, which is about 2000 ms. However, for a 3072 bit key and greater, there becomes a runway effect. The time difference ΔT between both algorithms is:

$$\Delta T_n = \overline{T}_{n_{RSA}} + \overline{T}_{n_{ECC}}$$

$$\Delta T_{1024} = 2158 - 2063 = 95\,ms$$

$$\Delta T_{3072} = 2849.7 - 2297 = 552.7\,ms$$

$$\Delta T_{15360} = 217288.4 - 3004.1 = 214284.3\,ms$$

The ranges of time intervals each algorithm are:

$$Range = \overline{T}_{highest} - \overline{T}_{lowest}$$

$$Range_{RSA} = 217288.4 - 2063.5 = 215224.9\,ms$$

$$Range_{ECC} = 3004.1 - 2158 = 846.1\,ms$$

The findings indicate that across 1024, 3072, and 15360 bit keys, RSA becomes increasingly and dramatically inefficient as compared to ECC. This is indicated by the steep increase in the range of time $\Delta T_{1024} < \Delta T_{3072} < \Delta T_{15360}$ and so RSA becomes increasingly and dramatically inefficient as compared to ECC as the key size increases.

According to graph 3, RSA constantly uses more memory than ECC.

Limitations

It was intended for the security tests to have more key sizes that are greater than 16 bits in order to better support the findings. However, brute forcing ECC for such key sizes is not made for regular computers. Doing so on a regular machine such as the one used for this Extended Essay will take extensive, immeasurable durations of time. These powerful computations are left for quantum computers that have sufficient power. RSA will start giving the same problem starting with 64 bit keys. Hence, the range of variables is not that wide.

Conclusion

The tests detailed in this Extended Essay serve as evidence to the superior build of Elliptic Curve Cryptosystems. They provide better security against attacks - thanks to its more intricate trap door design, and they are more cost-effective considering the savings made in memory and time as compared to RSA. Said savings may not be noteworthy to some, but they are crucial for businesses especially those that handle huge amounts of data.

This Extended Essay introduced two vastly employed cryptosystems: RSA, and Elliptic Curve Cryptography. It studied The mechanisms of both from when a message is first sent till it arrives to the receiver. It also studied the mathematical structure underpinning both algorithms to theoretically evaluate their feasibility. While RSA is secure, its use now requires the use of huge keys, This was later quantized and analyzed in the experiment that tested the security and costs in terms of memory and time. The conclusion drawn from this Extended Essay is that Elliptic Curve Cryptography is a superior option for cryptographic applications, due to the fact that it delivers better resistance against cyberattacks, and takes up fewer computational resources. This makes for the best return on investment for businesses that handle Big Data.

Bibliography

Adalier, M., & Teknik, A. (2015). *Efficient and Secure Elliptic Curve Cryptography Implementation of Curve P-256.*

Anderson, R.J. (1993). Practical RSA trapdoor. Electronics Letters, 29, 995-995.

Azaky, A. (2015, March 15). Ellipticcurvecryptography/SRC/ECC at master · Azaky/Ellipticcurvecryptography. GitHub. Retrieved from https://github.com/azaky/EllipticCurveCryptography/tree/master/src/ecc

Benaich, M. (2015, May 30). JavaDataStructures/ECC at master · Benaich/javadatastructures. GitHub. Retrieved from https://github.com/benaich/JavaDataStructures/tree/master/ECC

Chatzigiannakis, I., Pyrgelis, A., Spirakis, P., & Stamatiou, Y. (n.d.). *Elliptic curve based zero knowledge proofs and their applicability on resource constrained devices.* Retrieved from https://www.researchgate.net/publication/51915766_Elliptic_Curve_Based_Zero_Knowledge_Proofs_and_Their_Applicability_onResource_Constrained_Devices

Crépeau, C., Slakmon, A. (2003). Simple Backdoors for RSA Key Generation. In: Joye, M. (eds) Topics in Cryptology — CT-RSA 2003. CT-RSA 2003. Lecture Notes in Computer Science, vol 2612. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-36563-X_28

Cromwell, B. (2022, October 1). *Points on an elliptic curve. Bob Cromwell: Travel, Linux, Cybersecurity.* Retrieved from https://cromwell-intl.com/cybersecurity/elliptic-curve-cryptography/points-on-a-curve.html

Eckstein, L. (1996, November 21). *Brown University*. Retrieved from Modular exponentiation as a one-way function: http://cs.brown.edu/courses/cs007/oneway/node3.html

Gura, N., Patel, A., Wander, A., Hans, E., & Shantz, S. C. (n.d.). *Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs.* Sun Microsytem Laboratories.

Hellman, D., & Hellman, M. E. (1976). New Directions in Cryptography. *IEEE Transactions of Information Theory IT-22*.

K, S. (2020). Elliptic Curve Cryptography |Encryption and Decryption |ECC in Cryptography & Security. YouTube. Retrieved from https://www.youtube.com/watch?v=dhJX9kktijo

Lefton, P. (1991). Number Theory and Public-Key Cryptography. *The Mathematics Teacher 84(1)*, 54-62.

OldCurmudgeon. (2013, May 28). Faster prime factorization for huge BigIntegers in Java. Stack Overflow. Retrieved from https://stackoverflow.com/a/16802796

QuintessenceLabs. (2022, September 15). Breaking RSA encryption - an update on the state-of-the-art. QuintessenceLabs. Retrieved from https://www.quintessencelabs.com/blog/breaking-rsa-encryption-update-state-art#:~:text=It%20would%20take%20a%20classical,%E2%80%9Csafe%E2%80%9D%20from%20these%20attacks

Rai, D. (2018, March 10). RSA encryption and decryption in Java. devglan. Retrieved from https://www.devglan.com/java8/rsa-encryption-decryption-java

Rivlin, A. M., & Litan, R. E. (2001, December 1). *Brookings*. Retrieved from The Economy and the Internet: What Lies Ahead?: https://www.brookings.edu/research/the-economy-and-the-internet-what-lies-ahead/

Rivest, R. (1978). Remarks On A Proposed Cryptanalytic Attack On The M.I.T. Public-Key Cryptosystem. *Cryptologia*, 62-65.

Rivest, R., Shamir, A., & Adleman, L. (1977). *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.*

Sirajuddin, A. (2019). *The RSA Algorithm.*

Sutherland, A. (2017, February). *18.783 Elliptic Curves Lecture 1. 18.783 Elliptic Curves.* Boston; Massachusetts Institute of Technology.

Vogel, L. (2008). Java Performance - Memory and Runtime Analysis - Tutorial. vogella.com. Retrieved from https://www.vogella.com/tutorials/JavaPerformance/article.html

Wikimedia Commons. (2008). *Elliptic curve catalog. File:EllipticCurveCatalog.svg.* Retrieved from https://commons.wikimedia.org/wiki/File:EllipticCurveCatalog.svg.

Yang, H. (2013). *RsaKeyGenerator.java for RSA Key Generation*. RsaKeyGenerator.java for RSA key generation. Retrieved from http://www.herongyang.com/Cryptography/RSA-BigInteger-RsaKeyGenerator-java.html

Yang, H. (2022). *Same Point Addition on an Elliptic Curve.* Retrieved from http://www.herongyang.com/EC-Cryptography/Introduction-Same-Point-Addition-on-Elliptic-Curve.html

Appendix

RSA Security Test

```
/* RsaKeyGenerator.java
#- Copyright (c) 2013, HerongYang.com, All Rights Reserved.
*/
// Edited by the student for the Computer Science Extended
Essay
import java.math.BigInteger;
import java.util.*;
import java.io.*;
import java.lang.Math;

class RsaKeyGenerator {
   // Memoization of factors.
   static Map<BigInteger, List<BigInteger>> factors = new
HashMap<>();
   private static final BigInteger TWO =
BigInteger.ONE.add(BigInteger.ONE);
   BigInteger PHIn, e, d;

   public static void main(String[] args) {
       Random rnd = new Random();
       Scanner sc = new Scanner(System.in);

       // Gather the bitsize of p and / or q, which is half the
size of n.
       System.out.println("Bit size of p and q");
       int bitSize = sc.nextInt();

       // Generates a prime number with the bit size specified
by the first parameter
       BigInteger p = BigInteger.probablePrime(bitSize, rnd);
       BigInteger q = p.nextProbablePrime();

       // Yields the public modulus n.
       BigInteger n = p.multiply(q);

       System.out.println("Modulus: " + n);
       System.out.println("Key size: " + n.bitLength());

       // What follows is generated by the student for the
Extended Essay
```

```
        // Start of factoring the public modulus n
        // Measures the starting time
        long startTime = System.currentTimeMillis();

        // Factoring n
        factors(n, false);

        // After successfully factoring n, the security is
compromised.

        // Measures the end time
        long endTime = System.currentTimeMillis();

        // Calculates the difference, the elapsed time in
milliseconds
        long elapsedTime = endTime - startTime;

        System.out.println("Time elapsed (milliseconds) = " +
elapsedTime);
    }

    public static void factors(BigInteger n, boolean duplicates)
{
        // Have we done this one before?
        List<BigInteger> f = factors.get(n);

        if (f == null) {
            // Start empty.
            f = new ArrayList<>();

            // Check for duplicates.
            BigInteger last = BigInteger.ZERO;

            // Limit the range as far as possible.
            for (BigInteger i = TWO; i.compareTo(n.divide(i)) <=
0; i = i.add(BigInteger.ONE)) {
                // Can have multiple copies of the same factor.
                while (n.mod(i).equals(BigInteger.ZERO)) {
                    if (duplicates || !i.equals(last)) {
                        f.add(i);
                        last = i;
                    }
                    // Remove that factor.
                    n = n.divide(i);
                }
```

```
            }
            if (n.compareTo(BigInteger.ONE) > 0) {
                // Could be a residue.
                if (duplicates || n != last) {
                    f.add(n);
                }
            }
            // Memoize.
            factors.put(n, f);
        }
        System.out.println("Prime factors of the modulus n: 
"+f.toString());
    }
}
```

(Yang, 2013)

RSA Efficiency Test

```
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.math.BigInteger;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.RSAPrivateKeySpec;
import java.security.spec.RSAPublicKeySpec;
import java.util.Scanner;
import javax.crypto.Cipher;

/**
*
* @author Anuj
* Blog www.goldenpackagebyanuj.blogspot.com
* RSA - Encrypt Data using Public Key
```

```
* RSA - Descypt Data using Private Key
*/
public class RSAEncryptionDecryption {

   private static final String PUBLIC_KEY_FILE = "Public.key";
   private static final String PRIVATE_KEY_FILE =
"Private.key";

   public static void main(String[] args) throws IOException {

       long startTime = System.currentTimeMillis();

       try {
           System.out.println("-------GENRATE PUBLIC and
PRIVATE KEY-------------");
           KeyPairGenerator keyPairGenerator =
KeyPairGenerator.getInstance("RSA");
           Scanner sc = new Scanner(System.in);
           System.out.println("Gather key size");
           keyPairGenerator.initialize(sc.nextInt()); //1024
used for normal securities
           KeyPair keyPair =
keyPairGenerator.generateKeyPair();
           PublicKey publicKey = keyPair.getPublic();
           PrivateKey privateKey = keyPair.getPrivate();
           System.out.println("Public Key - " + publicKey);
           System.out.println("Private Key - " + privateKey);

           //Pullingout parameters which makes up Key
           System.out.println("\n------- PULLING OUT PARAMETERS
WHICH MAKES KEYPAIR----------\n");
           KeyFactory keyFactory =
KeyFactory.getInstance("RSA");
           RSAPublicKeySpec rsaPubKeySpec =
keyFactory.getKeySpec(publicKey, RSAPublicKeySpec.class);
           RSAPrivateKeySpec rsaPrivKeySpec =
keyFactory.getKeySpec(privateKey, RSAPrivateKeySpec.class);
           System.out.println("PubKey Modulus : " +
rsaPubKeySpec.getModulus());
           System.out.println("PubKey Exponent : " +
rsaPubKeySpec.getPublicExponent());
           System.out.println("PrivKey Modulus : " +
rsaPrivKeySpec.getModulus());
           System.out.println("PrivKey Exponent : " +
rsaPrivKeySpec.getPrivateExponent());
```

```
            //Share public key with other so they can encrypt
data and decrypt thoses using private key(Don't share with
Other)
            System.out.println("\n--------SAVING PUBLIC KEY AND
PRIVATE KEY TO FILES-------\n");
            RSAEncryptionDecryption rsaObj = new
RSAEncryptionDecryption();
            rsaObj.saveKeys(PUBLIC_KEY_FILE,
rsaPubKeySpec.getModulus(), rsaPubKeySpec.getPublicExponent());
            rsaObj.saveKeys(PRIVATE_KEY_FILE,
rsaPrivKeySpec.getModulus(),
rsaPrivKeySpec.getPrivateExponent());

            //Encrypt Data using Public Key
            byte[] encryptedData = rsaObj.encryptData("Anuj
Patel - Classified Information !");

            //Descypt Data using Private Key
            rsaObj.decryptData(encryptedData);

        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }catch (InvalidKeySpecException e) {
            e.printStackTrace();
        }

        // What follows is generated by the student for the
Extended Essay
        // Measures the end time
        long endTime = System.currentTimeMillis();
        // Calculates the difference, the elapsed time in
milliseconds (student-generated).
        long elapsedTime = endTime - startTime;
        System.out.println("Time Elapsed
(Milliseconds)"+elapsedTime);

        // Measure the allocated memory
        // Get the Java runtime
        Runtime runtime = Runtime.getRuntime();
        // Run the garbage collector
        runtime.gc();
        // Calculate the used memory
        long memory = runtime.totalMemory() -
runtime.freeMemory();
```

```java
        System.out.println("Used memory (Bytes): " + memory);
    }

    /**
     * Save Files
     * @param fileName
     * @param mod
     * @param exp
     * @throws IOException
     */
    private void saveKeys(String fileName,BigInteger
mod,BigInteger exp) throws IOException{
        FileOutputStream fos = null;
        ObjectOutputStream oos = null;

        try {
            System.out.println("Generating "+fileName + "...");
            fos = new FileOutputStream(fileName);
            oos = new ObjectOutputStream(new
BufferedOutputStream(fos));

            oos.writeObject(mod);
            oos.writeObject(exp);

            System.out.println(fileName + " generated
successfully");
        } catch (Exception e) {
            e.printStackTrace();
        }
        finally{
            if(oos != null){
                oos.close();

                if(fos != null){
                    fos.close();
                }
            }
        }
    }

    /**
     * Encrypt Data
     * @param data
     * @throws IOException
     */
```

```
    private byte[] encryptData(String data) throws IOException {
        System.out.println("\n----------------ENCRYPTION
STARTED------------");

        System.out.println("Data Before Encryption :" + data);
        byte[] dataToEncrypt = data.getBytes();
        byte[] encryptedData = null;
        try {
            PublicKey pubKey =
readPublicKeyFromFile(PUBLIC_KEY_FILE);
            Cipher cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.ENCRYPT_MODE, pubKey);
            encryptedData = cipher.doFinal(dataToEncrypt);
            System.out.println("Encryted Data: " +
encryptedData);

        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("----------------ENCRYPTION
COMPLETED------------");
        return encryptedData;
    }

    /**
     * Encrypt Data
     * @param data
     * @throws IOException
     */
    private void decryptData(byte[] data) throws IOException {
        System.out.println("\n----------------DECRYPTION
STARTED------------");
        byte[] descryptedData = null;

        try {
            PrivateKey privateKey =
readPrivateKeyFromFile(PRIVATE_KEY_FILE);
            Cipher cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.DECRYPT_MODE, privateKey);
            descryptedData = cipher.doFinal(data);
            System.out.println("Decrypted Data: " + new
String(descryptedData));

        } catch (Exception e) {
```

```
            e.printStackTrace();
        }

        System.out.println("----------------DECRYPTION
COMPLETED------------");
    }

    /**
     * read Public Key From File
     * @param fileName
     * @return PublicKey
     * @throws IOException
     */
    public PublicKey readPublicKeyFromFile(String fileName)
throws IOException{
        FileInputStream fis = null;
        ObjectInputStream ois = null;
        try {
            fis = new FileInputStream(new File(fileName));
            ois = new ObjectInputStream(fis);

            BigInteger modulus = (BigInteger) ois.readObject();
            BigInteger exponent = (BigInteger) ois.readObject();

            //Get Public Key
            RSAPublicKeySpec rsaPublicKeySpec = new
RSAPublicKeySpec(modulus, exponent);
            KeyFactory fact = KeyFactory.getInstance("RSA");
            PublicKey publicKey =
fact.generatePublic(rsaPublicKeySpec);

            return publicKey;

        } catch (Exception e) {
            e.printStackTrace();
        }
        finally{
            if(ois != null){
                ois.close();
                if(fis != null){
                    fis.close();
                }
            }
        }
        return null;
```

```
    }

    /**
     * read Public Key From File
     * @param fileName
     * @return
     * @throws IOException
     */
    public PrivateKey readPrivateKeyFromFile(String fileName)
throws IOException{
        FileInputStream fis = null;
        ObjectInputStream ois = null;
        try {
            fis = new FileInputStream(new File(fileName));
            ois = new ObjectInputStream(fis);

            BigInteger modulus = (BigInteger) ois.readObject();
            BigInteger exponent = (BigInteger) ois.readObject();

            //Get Private Key
            RSAPrivateKeySpec rsaPrivateKeySpec = new
RSAPrivateKeySpec(modulus, exponent);
            KeyFactory fact = KeyFactory.getInstance("RSA");
            PrivateKey privateKey =
fact.generatePrivate(rsaPrivateKeySpec);

            return privateKey;

        } catch (Exception e) {
            e.printStackTrace();
        }
        finally{
            if(ois != null){
                ois.close();
                if(fis != null){
                    fis.close();
                }
            }
        }
        return null;
    }
}
```
(Rai, 2018)

Elliptic Curve Cryptography Security Test

```
// CrackECC.java
// Edited by the student for the Computer Science Extended
Essay
import java.math.BigInteger;
import java.util.Random;
import java.util.Scanner;

/**
* This class implements El Gamal Public-Key Cryptography using
Elliptic Curve.
*
* This class also implements key-pair generation.
*
* @author Ahmad Zaky
*/
public class Main {
   public static final long AUXILIARY_CONSTANT_LONG = 1000;
   public static final BigInteger AUXILIARY_CONSTANT =
BigInteger.valueOf(AUXILIARY_CONSTANT_LONG);

   // The return type is changed from KeyPair to long to return
the cracking duration.
   public static long generate_and_crack_KeyPair(EllipticCurve
c, Random rnd, Scanner sc) throws Exception {
       // Randomly select the private key, such that it is
relatively prime to p
       BigInteger p = c.getP();
       BigInteger privateKey;

       // Gather the key bit size (student-generated)
       System.out.println("Enter the bit size of the private
key");
       do {
           privateKey = new BigInteger(sc.nextInt(), rnd);
       } while (privateKey.mod(p).compareTo(BigInteger.ZERO) ==
0);

       // Calculate the public key, k * g.
       // First, randomly generate g if it is not present in
the curve.
       ECPoint g = c.getBasePoint();
       if (g == null) {
           // Randomly generate g using Koblits method.
```

```
            // The starting value of x should be random.
            BigInteger x = new BigInteger(p.bitLength(), rnd);
            g = koblitzProbabilistic(c, x);
            c.setBasePoint(g);
        }
        ECPoint publicKey = c.multiply(g, privateKey);

        // Bruteforce the keypair by trying to find the public
point from the generator
        // Measures the starting time (student-generated).
        long startTime = System.currentTimeMillis();

        // Setup by marking the generator (student-generated)
        int i = 1;
        ECPoint tr = c.multiply(g,i);
        i++;

        // Loop that keeps adding G to itself till the public
point is reached (student-generated).
        // Counts the number of additions through i
(student-generated).
        while(tr.x.intValue() != publicKey.x.intValue() &&
tr.y.intValue() != publicKey.y.intValue()){
            tr = c.multiply(g, i);
            System.out.println(i);
            i++;
        }
        // What follows is generated by the student for the
Extended Essay
        // Measures the end time
        long endTime = System.currentTimeMillis();

        // Calculates the difference, the elapsed time in
milliseconds (student-generated).
        long elapsedTime = endTime - startTime;
        return elapsedTime;
    }

    private static ECPoint koblitzProbabilistic(EllipticCurve c,
BigInteger x) throws Exception {
        BigInteger p = c.getP();

        // throw an exception if p != 3 (mod 4)
        if (!p.testBit(0) || !p.testBit(1)) {
            throw new Exception("P should be 3 (mod 4)");
```

```
        }
        BigInteger pMinusOnePerTwo =
p.subtract(BigInteger.ONE).shiftRight(1);

        BigInteger tempX =
x.multiply(AUXILIARY_CONSTANT).mod(p);
        for (long k = 0; k < AUXILIARY_CONSTANT_LONG; ++k) {
            BigInteger newX = tempX.add(BigInteger.valueOf(k));

            // Calculates the rhs of the elliptic curve
equation, call it a
            BigInteger a = c.calculateRhs(newX);

            // Determine whether this value is a quadratic
residue modulo p
            // It is if and only if a ^ ((p - 1) / 2) = 1 (mod
p)
            if (a.modPow(pMinusOnePerTwo,
p).compareTo(BigInteger.ONE) == 0) {
                // We found it! Now, the solution is y = a ^ ((p
+ 1) / 4)
                BigInteger y =
a.modPow(p.add(BigInteger.ONE).shiftRight(2), p);
                return new ECPoint(newX.mod(p), y);
            }
        }

        // If we reach this point, then no point are found
within the limit.
        throw new Exception("No point found within the auxiliary
constant");
    }

    public static void main(String[] args) throws Exception {
        // using NIST_P_192 to test
        EllipticCurve c = EllipticCurve.NIST_P_192;
        Random rnd = new Random();
        Scanner sc = new Scanner(System.in);

        System.out.print("Time elapsed (ms):
"+generate_and_crack_KeyPair(c, rnd, sc));

    }
}
```

```
import java.math.BigInteger;

/**
* This class represents Elliptic Curve in Galois Field G(p).
The equation will
* be in form  y^2 = x^3 + ax + b (mod p), for which a and b
satisfy
* 4a^3 + 27b^2 != 0 (mod p).
*
* A point in the elliptic curve will be represented as a pair
of BigInteger,
* which is represented as ECPoint class.
*
* This class implements some very basic operations of Points in
the elliptic
* curve, which are addition, multiplication (scalar), and
subtraction.
*
* @author Ahmad Zaky
*/
public class EllipticCurve {

   // The three parameters of the elliptic curve equation.
   private BigInteger a;
   private BigInteger b;
   private BigInteger p;

   // Optional attribute, the base point g.
   private ECPoint g = null;

   // some BigInteger constants that might help us in some
calculations
   private static BigInteger THREE = new BigInteger("3");

   public EllipticCurve(BigInteger a, BigInteger b, BigInteger
p) {
       this.a = a;
       this.b = b;
       this.p = p;
   }

   public EllipticCurve(BigInteger a, BigInteger b, BigInteger
p, ECPoint g) {
       this.a = a;
       this.b = b;
```

```
        this.p = p;
        this.g = g;
    }

    public EllipticCurve(long a, long b, long p) {
        this.a = BigInteger.valueOf(a);
        this.b = BigInteger.valueOf(b);
        this.p = BigInteger.valueOf(p);
    }

    public EllipticCurve(long a, long b, long p, ECPoint g) {
        this.a = BigInteger.valueOf(a);
        this.b = BigInteger.valueOf(b);
        this.p = BigInteger.valueOf(p);
        this.g = g;
    }

    public ECPoint getBasePoint() {
        return g;
    }

    public void setBasePoint(ECPoint g) {
        this.g = g;
    }

    public BigInteger getA() {
        return a;
    }

    public BigInteger getB() {
        return b;
    }

    public BigInteger getP() {
        return p;
    }

    // We provide some standard curves
    // Source:
http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.
pdf

    public static final EllipticCurve NIST_P_192 = new
EllipticCurve(
            new BigInteger("-3"),
```

```
            new
BigInteger("64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1",
16),
            new
BigInteger("6277101735386680763835789423207666416083908700390324961279"),
            new ECPoint(
                    new
BigInteger("188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012",
16),
                    new
BigInteger("07192b95ffc8da78631011ed6b24cdd573f977a11e794811",
16)
            )
    );

    /**
     * Warning: p = 1 (mod 4), cannot be used throughout the
algorithm.
     */
    public static final EllipticCurve NIST_P_224 = new
EllipticCurve(
            new BigInteger("-3"),
            new
BigInteger("b4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4", 16),
            new
BigInteger("26959946667150639794667015087019630673557916260026308143510066298881"),
            new ECPoint(
                    new
BigInteger("b70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21", 16),
                    new
BigInteger("bd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34", 16)
            )
    );

    public static final EllipticCurve NIST_P_256 = new
EllipticCurve(
            new BigInteger("-3"),
            new
BigInteger("5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b", 16),
```

```
           new
BigInteger("115792089210356248762697446949407573530086143415290
314195533631308867097853951"),
           new ECPoint(
                   new
BigInteger("6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a
13945d898c296", 16),
                   new
BigInteger("4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb
6406837bf51f5", 16)
           )
   );

   public static final EllipticCurve NIST_P_384 = new
EllipticCurve(
           new BigInteger("-3"),
           new
BigInteger("b3312fa7e23ee7e4988e056be3f82d19181d9c6efe814112031
4088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef", 16),
           new
BigInteger("394020061963944792122790401001436138050797392704654
466679482934042457217714968703290472660882589380018616069731123
19"),
           new ECPoint(
                   new
BigInteger("aa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f
741e082542a385502f25dbf55296c3a545e3872760ab7", 16),
                   new
BigInteger("3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9d
a3113b5f0b8c00a60b1ce1d7e819d7a431d7c90ea0e5f", 16)
           )
   );

   public static final EllipticCurve NIST_P_521 = new
EllipticCurve(
           new BigInteger("-3"),
           new
BigInteger("051953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3
b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f
1ef451fd46b503f00", 16),
           new
BigInteger("686479766013060971498190079908139321726943530014330
540939446345918554318339765605212255964066145455497729631139148
0858037121987999716643812574028291115057151"),
           new ECPoint(
```

```
                   new
BigInteger("c6858e06b70404e9cd9e3ecb662395b4429c648139053fb521f
828af606b4d3dbaa14b5e77efe75928fe1dc127a2ffa8de3348b3c1856a429b
f97e7e31c2e5bd66", 16),
                   new
BigInteger("11839296a789a3bc0045c8a5fb42c7d1bd998f54449579b4468
17afbd17273e662c97ee72995ef42640c550b9013fad0761353c7086a272c24
088be94769fd16650", 16)
           )
   );

   /**
    * This method will check whether a point belong to this
curve or not.
    */
   public boolean isPointInsideCurve(ECPoint point) {
       if (point.isPointOfInfinity()) return true;

       return
point.x.multiply(point.x).mod(p).add(a).multiply(point.x).add(b
)

.mod(p).subtract(point.y.multiply(point.y)).mod(p)
               .compareTo(BigInteger.ZERO) == 0;
   }

   /**
    * Add two points. The result of this addition is the
reflection of the
    * intersection of the line formed by the two points to the
same curve with
    * respect to the x-axis. The line is the tangent when the
two points equal.
    *
    * The result will be point of infinity when the line is
parallel to the
    * y-axis.
    *
    * If one of them is point of infinity, then the other will
be returned.
    *
    * @param p1
    * @param p2
    * @return
    */
```

```
    public ECPoint add(ECPoint p1, ECPoint p2) {
        if (p1 == null || p2 == null) return null;

        if (p1.isPointOfInfinity()) {
            return new ECPoint(p2);
        } else if (p2.isPointOfInfinity()) {
            return new ECPoint(p1);
        }

        // The lambda (the slope of the line formed by the two
points) are
        // different when the two points are the same.
        BigInteger lambda;
        if
(p1.x.subtract(p2.x).mod(p).compareTo(BigInteger.ZERO) == 0) {
            if
(p1.y.subtract(p2.y).mod(p).compareTo(BigInteger.ZERO) == 0) {
                // lambda = (3x1^2 + a) / (2y1)
                BigInteger nom =
p1.x.multiply(p1.x).multiply(THREE).add(a);
                BigInteger den = p1.y.add(p1.y);
                lambda = nom.multiply(den.modInverse(p));
            } else {
                // lambda = infinity
                return ECPoint.INFINTIY;
            }
        } else {
            // lambda = (y2 - y1) / (x2 - x1)
            BigInteger nom = p2.y.subtract(p1.y);
            BigInteger den = p2.x.subtract(p1.x);
            lambda = nom.multiply(den.modInverse(p));
        }

        // Now the easy part:
        // The result is (lambda^2 - x1 - y1, lambda(x2 - xr) -
yp)
        BigInteger xr =
lambda.multiply(lambda).subtract(p1.x).subtract(p2.x).mod(p);
        BigInteger yr =
lambda.multiply(p1.x.subtract(xr)).subtract(p1.y).mod(p);
        return new ECPoint(xr, yr);
    }

    /**
     * Subtract two points, according to this equation: p1 - p2
```

```
= p1 + (-p2),
     * where -p2 is the reflection of p2 with respect to the
x-axis.
     *
     * @param p1
     * @param p2
     * @return
     */
    public ECPoint subtract(ECPoint p1, ECPoint p2) {
        if (p1 == null || p2 == null) return null;

        return add(p1, p2.negate());
    }

    /**
     * Multiply p1 to a scalar n. That is, perform addition n
times. The
     * following method implements divide and conquer approach.
     *
     * @param p1
     * @param n
     * @return
     */
    public ECPoint multiply(ECPoint p1, BigInteger n) {
        if (p1.isPointOfInfinity()) {
            return ECPoint.INFINTIY;
        }

        ECPoint result = ECPoint.INFINTIY;
        int bitLength = n.bitLength();
        for (int i = bitLength - 1; i >= 0; --i) {
            result = add(result, result);
            if (n.testBit(i)) {
                result = add(result, p1);
            }
        }

        return result;
    }

    public ECPoint multiply(ECPoint p1, long n) {
        return multiply(p1, BigInteger.valueOf(n));
    }

    /**
```

```
     * Calculate the right hand side of the equation.
     *
     * @param x
     * @return
     */
    public BigInteger calculateRhs(BigInteger x) {
        return
x.multiply(x).mod(p).add(a).multiply(x).add(b).mod(p);
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Check whether the standard curves' base points lie on
the curve
        System.out.println("NIST_P_192: " +
EllipticCurve.NIST_P_192.isPointInsideCurve(EllipticCurve.NIST_
P_192.getBasePoint()));
        System.out.println("NIST_P_224: " +
EllipticCurve.NIST_P_224.isPointInsideCurve(EllipticCurve.NIST_
P_224.getBasePoint()));
        System.out.println("NIST_P_256: " +
EllipticCurve.NIST_P_256.isPointInsideCurve(EllipticCurve.NIST_
P_256.getBasePoint()));
        System.out.println("NIST_P_384: " +
EllipticCurve.NIST_P_384.isPointInsideCurve(EllipticCurve.NIST_
P_384.getBasePoint()));
        System.out.println("NIST_P_521: " +
EllipticCurve.NIST_P_521.isPointInsideCurve(EllipticCurve.NIST_
P_521.getBasePoint()));

        for (int i = 0; i < 20; ++i) {
            System.out.println("NIST_P_521 x " + i + " = " +
EllipticCurve.NIST_P_521.multiply(EllipticCurve.NIST_P_521.getB
asePoint(), i).toString(16));
        }

        // This computes (2, 4) + (5, 9) in y^2 = x^3 + x + 6
mod 11
        EllipticCurve e = new EllipticCurve(1, 6, 11);
        ECPoint p = new ECPoint(3, 5);
        ECPoint q = new ECPoint(5, 9);

        System.out.println(p + " + " + q + " = " + e.add(p, q));
```

```java
        for (int i = 0; i < 20; ++i) {
            System.out.println(p + " x " + i + " = " +
e.multiply(p, i));
        }
    }

}
```

```java
import java.math.BigInteger;

/**
* This class will represent a point inside an elliptic curve.
*
* @author Ahmad Zaky
*/
public class ECPoint {
    public BigInteger x;
    public BigInteger y;
    private boolean pointOfInfinity;

    public ECPoint() {
        this.x = this.y = BigInteger.ZERO;
        this.pointOfInfinity = false;
    }

    public ECPoint(BigInteger x, BigInteger y) {
        this.x = x;
        this.y = y;
        this.pointOfInfinity = false;
    }

    public ECPoint(long x, long y) {
        this.x = BigInteger.valueOf(x);
        this.y = BigInteger.valueOf(y);
        this.pointOfInfinity = false;
    }

    public ECPoint(ECPoint p) {
        this.x = p.x;
        this.y = p.y;
        this.pointOfInfinity = p.pointOfInfinity;
    }

    public boolean equals(ECPoint point) {
        if (point == null) return false;
```

```java
        if (this.pointOfInfinity == point.pointOfInfinity)
return true;

        return (this.x.compareTo(point.x) |
this.y.compareTo(point.y)) == 0;
    }

    public boolean isPointOfInfinity() {
        return pointOfInfinity;
    }

    public ECPoint negate() {
        if (isPointOfInfinity()) {
            return INFINTIY;
        } else {
            return new ECPoint(x, y.negate());
        }
    }

    private static ECPoint infinity() {
        ECPoint point = new ECPoint();
        point.pointOfInfinity = true;
        return point;
    }

    public static final ECPoint INFINTIY = infinity();

    @Override
    public String toString() {
        if (isPointOfInfinity()) {
            return "INFINITY";
        } else {
            return "(" + x.toString() + ", " + y.toString() +
")";
        }
    }

    public String toString(int radix) {
        if (isPointOfInfinity()) {
            return "INFINITY";
        } else {
            return "(" + x.toString(radix) + ", " +
y.toString(radix) + ")";
        }
```

```
    }
}
```

```
import java.math.BigInteger;
import java.security.PrivateKey;
import java.security.PublicKey;

/**
* The class will contain a pair of public key and private key.
*
* @author Ahmad Zaky
*/
public class KeyPair {
   private BigInteger publicKey;
   private BigInteger privateKey;

   public KeyPair(BigInteger publicKey, BigInteger privateKey)
{
       this.publicKey = publicKey;
       this.privateKey = privateKey;
   }
}
```

(Azaky, 2015)

Elliptic Curve Cryptography Efficiency Test

```
import java.math.BigInteger;
import java.util.HashMap;
import java.util.Random;
import java.util.Scanner;

public class ECC {

   public static int PAD = 5;
   public static final Random r = new Random();

   private HashMap<Point, Integer> pointTable;
   private HashMap<Integer, Point> charTable;

   private Encoder mEncoder;
   private Decoder mDecoder;

   public ECC(EllipticCurve c) {
       initCodeTable(c);
```

```
        this.mEncoder = new Encoder(charTable);
        this.mDecoder = new Decoder(pointTable);
    }

    public static Random getRandom() {
        return r;
    }

    private int[] encrypt(String msg, PublicKey key) {
        EllipticCurve c = key.getCurve();
        Point g = c.getBasePoint();
        Point publicKey = key.getKey();
        BigInteger p = c.getP();
        int numBits = p.bitLength();
        BigInteger k;
        do {
            k = new BigInteger(numBits, getRandom());
        } while (k.mod(p).compareTo(BigInteger.ZERO) == 0);
        Point sharedSecret = c.multiply(publicKey, k);

        Point keyHint = c.multiply(g, k); // key to send

        System.out.println("----------------- Encryption process
-----------------");
        System.out.println(c);
        System.out.println("Mesage to encrypt, m = " + msg);
        System.out.println("Bob's public key, Pb = " +
publicKey);
        System.out.println("Alice's private key, k = " + k);
        System.out.println("The ecryption key, sharedSecret = k
* Pb = " + sharedSecret);
        System.out.println("The hint to compute sharedSecret for
bob, keyHint = " + keyHint);

        Matrix mMatrix = mEncoder.encode(msg);
        mMatrix.performAddition(Helpers.toBinary(sharedSecret));
        System.out.println("sharedSecret binary format :");
        Helpers.print(Helpers.toBinary(sharedSecret));
        System.out.println("4) encrypt the matrix with
sharedSecret (code addition)");
        System.out.println(mMatrix);
        return mMatrix.toArray(Helpers.toBinary(keyHint));
    }

    private String decrypt(int[] cipherText, PrivateKey key) {
```

```
        EllipticCurve c = key.getCurve();
        Point g = c.getBasePoint();
        BigInteger privateKey = key.getKey();
        BigInteger p = c.getP();

        Point keyHint = Point.make(cipherText);
        Point sharedSecret = c.multiply(keyHint, privateKey);

        System.out.println("\n----------------- Decryption
process -----------------");
        System.out.println("1) Bob receive this :");
        Helpers.print(cipherText);
        System.out.println("");
        System.out.println("2) Extract keyhint and the matrix
C");
        System.out.println("KeyHint = "+keyHint);

        //get the decypted matrix
        Matrix mMatrix = Matrix.make(cipherText);
        System.out.println("C = ");
        System.out.println(mMatrix);
        //substract the key form the matrix

mMatrix.performSubstraction(Helpers.toBinary(sharedSecret));
        System.out.println("Matrix after substraction");
        System.out.println(mMatrix);
        //decode the matrix
        System.out.println("3) Reverse Matrix Scrambling");
        return mDecoder.decode(mMatrix);
    }

    /**
     * Generate a random key-pair, given the elliptic curve
being used.
     */
    public static KeyPair generateKeyPair(EllipticCurve c) {
        // Randomly select the private key, such that it is
relatively prime to p
        BigInteger p = c.getP();
        BigInteger privateKey;
        System.out.println("Gather key size");
        Scanner sc = new Scanner(System.in);
        do {
            privateKey = new BigInteger(sc.nextInt(),
getRandom());
```

```
        } while (privateKey.mod(p).compareTo(BigInteger.ZERO) ==
0);

        // Calculate the public key, k * g.
        Point g = c.getBasePoint();
        Point publicKey = c.multiply(g, privateKey);

        return new KeyPair(
                new PublicKey(c, publicKey),
                new PrivateKey(c, privateKey)
        );
    }

    public final void initCodeTable(EllipticCurve curve) {
        charTable = new HashMap<>();
        pointTable = new HashMap<>();
        Point p = curve.getBasePoint();
        for (int i = 1; i < 27; i++) {
            do {
                p = curve.multiply(curve.getBasePoint(), i);
            } while (p.isInfinity());
            charTable.put(i + 96, p); // 0 here refers to char
97 witch is a
        }
        //special characters
        charTable.put(32, Point.getInfinity()); //space
        int[] codeAscii = new int[]{10, 13, 39, 40, 41, 44, 46,
58, 59};
        for (int i : codeAscii) {
            p = curve.add(p, curve.getBasePoint());
            charTable.put(i, p);
        }

        //populate the points symbol table
        for (Integer key : charTable.keySet()) {
            pointTable.put(charTable.get(key), key);
        }
    }

    public void displayCodeTable() {
        System.out.println("------ Code Table -------");
        charTable.forEach((cle, val) -> {
            System.out.println((char) cle.intValue() + " -> " +
val);
        });
```

```java
    }

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        EllipticCurve c = new EllipticCurve(4, 20, 29, new
Point(1, 5));
        ECC ecc = new ECC(c);
        ecc.displayCodeTable();
        String msg = "i understood the importance in principle
of public key cryptography, but it is all moved much faster
than i expected i did not expect it to be a mainstay of
advanced communications technology";
        msg = "hi there";
        // generate pair of keys
        KeyPair keys = generateKeyPair(c);
        // encrypt the msg
        int[] cipherText = ecc.encrypt(msg,
keys.getPublicKey());
        System.out.println("5) Alice send this to Bob:");
        Helpers.print(cipherText);

        // decrypt the result
        String plainText = ecc.decrypt(cipherText,
keys.getPrivateKey());
        System.out.println("\n5) Translate each point to a
carracter");

        //System.out.println("Cipher : ");
        //Helpers.print(cipherText);
        System.out.println("Plain text : \n" + plainText);

        // What follows is generated by the student for the
Extended Essay
        // Measures the end time
        long endTime = System.currentTimeMillis();
        // Calculates the difference, the elapsed time in
milliseconds (student-generated).
        long elapsedTime = endTime - startTime;
        System.out.println("Time Elapsed (Milliseconds):
"+elapsedTime);

        // Measure the allocated memory
        // Get the Java runtime
        Runtime runtime = Runtime.getRuntime();
        // Run the garbage collector
```

```java
        runtime.gc();
        // Calculate the used memory
        long memory = runtime.totalMemory() -
runtime.freeMemory();
        System.out.println("Used memory (Bytes): " + memory);
    }
}
```

```java
import java.math.BigInteger;

public class EllipticCurve {

    // The three parameters of the elliptic curve equation.
    private BigInteger a;
    private BigInteger b;
    private BigInteger p;

    // Optional attribute, the base point g.
    private Point basePoint = null;
    // some BigInteger constants that might help us in some
calculations
    private static BigInteger THREE = new BigInteger("3");

    public EllipticCurve(BigInteger a, BigInteger b, BigInteger
p, Point g) {
        this.a = a;
        this.b = b;
        this.p = p;
        this.basePoint = g;
    }

    public EllipticCurve(BigInteger a, BigInteger b, BigInteger
p) {
        this(a, b, p, null);
    }

    public EllipticCurve(long a, long b, long p, Point g) {
        this(BigInteger.valueOf(a), BigInteger.valueOf(b),
BigInteger.valueOf(p), g);
    }

    public EllipticCurve(long a, long b, long p) {
        this(a, b, p, null);
    }
```

```
    public BigInteger getA() {
        return a;
    }

    public BigInteger getB() {
        return b;
    }

    public BigInteger getP() {
        return p;
    }

    public Point getBasePoint() {
        return basePoint;
    }
    public void setBasePoint(Point p) {
        basePoint = p;
    }

    /**
     * This method will check whether a point belong to this
curve or not.
     */
    public boolean contains(Point point) {
        if (point.isInfinity()) {
            return true;
        }

        return
point.getX().multiply(point.getX()).mod(p).add(a).multiply(poin
t.getX()).add(b)

.mod(p).subtract(point.getY().multiply(point.getY())).mod(p)
               .compareTo(BigInteger.ZERO) == 0;
    }

    /**
     * add two point
     */
    public Point add(Point p1, Point p2) {
        if (p1 == null || p2 == null) {
            return null;
        }

        if (p1.isInfinity()) {
```

```
            return new Point(p2);
        } else if (p2.isInfinity()) {
            return new Point(p1);
        }

        // The lambda (the slope of the line formed by the two
points) is different when the two points are the same.
        BigInteger lambda;
        if
(p1.getX().subtract(p2.getX()).mod(p).compareTo(BigInteger.ZERO
) == 0) {
            if
(p1.getY().subtract(p2.getY()).mod(p).compareTo(BigInteger.ZERO
) == 0) {
                // lambda = (3x1^2 + a) / (2y1)
                BigInteger nom =
p1.getX().multiply(p1.getX()).multiply(THREE).add(a);
                BigInteger den = p1.getY().add(p1.getY());
                lambda = nom.multiply(den.modInverse(p));
            } else {
                // lambda = infinity
                return Point.getInfinity();
            }
        } else {
            // lambda = (y2 - y1) / (x2 - x1)
            BigInteger nom = p2.getY().subtract(p1.getY());
            BigInteger den = p2.getX().subtract(p1.getX());
            lambda = nom.multiply(den.modInverse(p));
        }

        // Now the easy part:
        // The result is (lambda^2 - x1 - y1, lambda(x2 - xr) -
yp)
        BigInteger xr =
lambda.multiply(lambda).subtract(p1.getX()).subtract(p2.getX())
.mod(p);
        BigInteger yr =
lambda.multiply(p1.getX().subtract(xr)).subtract(p1.getY()).mod
(p);
        return new Point(xr, yr);
    }

    /**
     * Subtract two points, according to this equation: p1 - p2
= p1 + (-p2),
```

```
     */
    public Point subtract(Point p1, Point p2) {
        if (p1 == null || p2 == null) {
            return null;
        }
        return add(p1, p2.negate());
    }

    /**
     * Multiply p1 to a scalar n. That is, perform addition n
times.
     */
    public Point multiply(Point p1, BigInteger n) {
        if (p1.isInfinity()) {
            return Point.getInfinity();
        }

        Point result = Point.getInfinity();
        int bitLength = n.bitLength();
        for (int i = bitLength - 1; i >= 0; --i) {
            result = add(result, result);
            if (n.testBit(i)) {
                result = add(result, p1);
            }
        }

        return result;
    }

    public Point multiply(Point p1, long n) {
        return multiply(p1, BigInteger.valueOf(n));
    }

    /**
     * Calculate the right hand side of the equation.
     */
    public BigInteger calculateRhs(BigInteger x) {
        return
x.multiply(x).mod(p).add(a).multiply(x).add(b).mod(p);
    }

    public static void main(String[] args) {
        // This computes (2, 4) + (5, 9) in y^2 = x^3 + x + 6
mod 11
        EllipticCurve e = new EllipticCurve(4, 20, 29);
```

```
        Point p = new Point(1, 5);
        Point q = new Point(5, 9);

        System.out.println(p + " + " + q + " = " + e.add(p, q));
        for (int i = 0; i < 20; ++i) {
            System.out.println(p + " x " + i + " = " +
e.multiply(p, i));
        }
    }

    @Override
    public String toString() {
        return "EllipticCurve EC : " + "y^2 = x^3 + " + a + "x +
" + b + "c mod " + p + "\nGenerator point G = " + basePoint +
'}';
    }

}
```

```
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class Encoder {

    private HashMap<Integer, Point> charTable;

    public Encoder(HashMap<Integer, Point> charTable) {
        this.charTable = charTable;
    }

    public Matrix encode(String plainText) {
        Matrix mMatrix = createMatrix(plainText);
        System.out.println("\n2) Convert the list of points to a
binary Matrix");
        System.out.println(mMatrix);
        System.out.println("\n3) Matrix Scrambling");
        int w = new BigInteger(ECC.PAD,
ECC.getRandom()).intValue();
        int[] bits =
Helpers.toBinary(ECC.getRandom().nextInt(1024), ECC.PAD * 2);
        System.out.println("number of transformations, w = " +
w);
        System.out.println("Random sequence of bits, Bits = ");
```

```java
        Helpers.print(bits);
        int bit, i = 0;
        do {
            bit = bits[i];
            if (bit == 0) {
                mMatrix.scramble(true);
            } else {
                mMatrix.scramble(false);
            }
            if (i == bits.length - 1) {
                i = 0;
            }else{
                i++;
            }
            //System.out.println("scrambling...");
            System.out.println(mMatrix);
            w--;
        } while (w > 0);
        return mMatrix;
    }

    private Matrix createMatrix(String plainText) {
        List<Point> pList = new ArrayList<>();
        for (Character c : plainText.toCharArray()) {
            Point p = charTable.get((int) c.charValue());
            pList.add(p);
        }
        System.out.println("\n1) Convert m to a list of
Points");
        pList.stream().forEach(System.out::print);
        System.out.println("");
        List<Integer> bList = new ArrayList<>();
        for (Point p : pList) {
            String str = Helpers.toBinary(p.getX()) + "" +
Helpers.toBinary(p.getY());
            for (int i = 0; i < str.length(); i++) {
                bList.add((str.charAt(i) == '0') ? 0 : 1);
            }
        }
        return Helpers.listToMatrix(bList);
    }

}
```

```java
import java.util.HashMap;
```

```java
import java.util.List;

public class Decoder {

    public static int PAD = 5;
    private HashMap<Point, Integer> pointTable;

    public Decoder(HashMap<Point, Integer> pointTable) {
        this.pointTable = pointTable;
    }

    public String decode(Matrix A) {
        List<String> subKeys = Helpers.getSubKeys();
        for (String subKey : subKeys) {
            String[] array = subKey.split("\\/");
            String transformation = array[0];
            int op = Integer.parseInt(array[1]);
            int a = Integer.parseInt(array[2]);
            int b = Integer.parseInt(array[3]);
            int c = Integer.parseInt(array[4]);
            int d = Integer.parseInt(array[5]);
            if (transformation.equals("R")) {
                A.reverseRowTrasformation(a, c, d, op);
                A.reverseRowTrasformation(b, c, d, op);
            } else {
                A.reverseColumnTrasformation(a, c, d, op);
                A.reverseColumnTrasformation(b, c, d, op);
            }
            System.out.println("sub-key : " + subKey);
            System.out.println(A);
        }
        return getPlainText(A);
    }

    private String getPlainText(Matrix A) {
        String plainText = "";
        System.out.println("4) Convert the matrix M to a list of
points");
        A.toPoints().stream().forEach(System.out::print);
        System.out.println("");

        for (Point p : A.toPoints()) {
            if (pointTable.get(p) != null) {
                int asciCode = pointTable.get(p);
                plainText += Character.toString((char)
```

```
asciCode);
            } else {
                plaintText += "$";
            }
        }
        return plaintText;
    }
}
```

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Helpers {

    static final File SUB_KEYS_FILE = new File("subkey");

    static int[] toBinary(int n, int base) {
        final int[] bits = new int[base];
        for (int i = 0; i < base; i++) {
            bits[i] = n >> i & 1;
        }
        return bits;
    }

    static String toBinary(BigInteger x) {
        String ret = "";
        for (int b : toBinary(x.intValue(), ECC.PAD)) {
            ret = b + ret;
        }
        return ret;
    }

    static String[] toBinary(Point p) {
        String[] tab = new String[ECC.PAD];
        String str = toBinary(p.getX()) + "" +
```

```
toBinary(p.getY());
        for (int i = 0; i < str.length(); i = i + 2) {
            tab[i / 2] = str.charAt(i) + "" + str.charAt(i + 1);
        }
        return tab;
    }

    static Matrix listToMatrix(List<Integer> list) {
        int n, m, row, col;
        n = ECC.PAD;
        m = list.size() / (2 * n);
        String[][] bits = new String[n][m];
        for (int i = 0; i < list.size(); i = i + 2) {
            row = i / 2 % n;
            col = i / 2 / n;
            bits[row][col] = list.get(i) + "" + list.get(i + 1);
        }
        return Matrix.make(bits);
    }

    static void saveSubKey(String t, int op, int a, int b, int
c, int d) {
        try {
            BufferedWriter writer;
            writer = new BufferedWriter(new
FileWriter(SUB_KEYS_FILE, true));
            writer.append(t + "/" + op + "/" + a + "/" + b + "/"
+ c + "/" + d);
            writer.newLine();
            writer.close();
        } catch (IOException ex) {

Logger.getLogger(Helpers.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

    static void saveEncryptedMsg(char t, int op, int a, int b,
int c, int d) {
        try {
            BufferedWriter writer;
            writer = new BufferedWriter(new
FileWriter(SUB_KEYS_FILE, true));
            writer.append(t + "/" + op + "/" + a + "/" + b + "/"
+ c + "/" + d);
```

```
            writer.newLine();
            writer.close();
        } catch (IOException ex) {

Logger.getLogger(Helpers.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

    static List<String> getSubKeys() {
        List<String> keys = new ArrayList<>();
        try {
            BufferedReader reader;
            reader = new BufferedReader(new
FileReader(SUB_KEYS_FILE));
            String line;
            while ((line = reader.readLine()) != null) {
                keys.add(line);
            }
            reader.close();
        } catch (IOException ex) {

Logger.getLogger(Helpers.class.getName()).log(Level.SEVERE,
null, ex);
        }
        SUB_KEYS_FILE.deleteOnExit();
        Collections.reverse(keys);
        return keys;
    }

    static void print(String[] tab) {
        for (String tab1 : tab) {
            System.out.println(tab1 + " ");
        }
    }

    static void print(int[] tab) {
        byte b;
        for (int i : tab) {
            b = (byte) i;
            System.out.print(b);
        }
        System.out.println("");
    }
```

```java
    static int getNotEqualTo(int a, int limit) {
        int b;
        do {
            b = ECC.getRandom().nextInt(limit);
        } while (b == a);
        return b;
    }

    static void main(String[] args) {
        for (int a : toBinary(15, 5)) {
            System.out.print(a);
        }
        System.out.println(" :" + Character.toString((char) 32)
+ ":");
    }
}
```

```java
public class KeyPair {

    private PublicKey publicKey;
    private PrivateKey privateKey;

    public KeyPair(PublicKey publicKey, PrivateKey privateKey) {
        this.publicKey = publicKey;
        this.privateKey = privateKey;
    }

    public PublicKey getPublicKey() {
        return publicKey;
    }

    public void setPublicKey(PublicKey publicKey) {
        this.publicKey = publicKey;
    }

    public PrivateKey getPrivateKey() {
        return privateKey;
    }

    public void setPrivateKey(PrivateKey privateKey) {
        this.privateKey = privateKey;
    }
}
```

```java
import java.math.BigInteger;
```

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Matrix {

   private final String[][] data;
   private final int rows;
   private final int columns;

   public Matrix(String[][] data) {
       this.data = data;
       this.rows = data.length;
       this.columns = data[0].length;
   }

   public int countRows() {
       return rows;
   }

   public int countColumns() {
       return columns;
   }

   public String[][] getData() {
       return data;
   }

   public String getData(int r, int c) {
       return data[r][c];
   }

   public static Matrix make(String[][] data) {
       return new Matrix(data);
   }

   public static Matrix make(int[] array) {
       int keyLength = 2 * ECC.PAD;
       Integer[] data = new Integer[(array.length -
keyLength)];
       // get the matrix
       for (int i = 0; i < data.length; i++) {
           data[i] = array[i + keyLength];
       }
       Matrix M = Helpers.listToMatrix(Arrays.asList(data));
```

```
        return M;
    }

    public void scramble(boolean type) {
        int n, m, row1, row2, col1, col2, x1, x2, op;
        n = data[0].length;
        m = data.length;
        row1 = ECC.getRandom().nextInt(m);
        row2 = Helpers.getNotEqualTo(row1, m);
        col1 = ECC.getRandom().nextInt(n);
        col2 = Helpers.getNotEqualTo(col1, m);
        op = new BigInteger(2, ECC.getRandom()).intValue();
//operations:
        if (type) {
            x1 = Integer.min(col1, col2); // first index
            x2 = Integer.max(col1, col2); // last index
            rowTrasformation(row1, x1, x2, op);
            rowTrasformation(row2, x1, x2, op);
            log("R", op, row1, row2, x1, x2);
        } else {
            x1 = Integer.min(row1, row2);
            x2 = Integer.max(row1, row2);
            columnTrasformation(col1, x1, x2, op);
            columnTrasformation(col2, x1, x2, op);
            log("C", op, col1, col2, x1, x2);
        }
    }

    public void rowTrasformation(int r1, int x1, int x2, int op)
{
        if (op == 0) {
            circularLeftShift(r1, x1, x2);
        } else if (op == 1) {
            circularRightShift(r1, x1, x2);
        } else {
            reverseRow(r1, x1, x2);
        }
    }

    public void columnTrasformation(int c1, int x1, int x2, int
op) {
        if (op == 0) {
            circularUpwardShift(c1, x1, x2);
        } else if (op == 1) {
```

```
            circularDownwardShift(c1, x1, x2);
        } else {
            reverseColumn(c1, x1, x2);
        }
    }

    public void reverseRowTrasformation(int r1, int x1, int x2,
int op) {
        if (op == 0) {
            op = 1;
        } else if (op == 1) {
            op = 0;
        }
        rowTrasformation(r1, x1, x2, op);
    }

    public void reverseColumnTrasformation(int c1, int x1, int
x2, int op) {
        if (op == 0) {
            op = 1;
        } else if (op == 1) {
            op = 0;
        }
        columnTrasformation(c1, x1, x2, op);
    }

    public void circularLeftShift(int row, int c1, int c2) {
        String tmp = data[row][c1];
        for (int i = c1; i < c2; i++) {
            data[row][i] = data[row][i + 1];
        }
        data[row][c2] = tmp;
    }

    public void circularRightShift(int row, int c1, int c2) {
        String tmp = data[row][c2];
        for (int i = c2; i > c1; i--) {
            data[row][i] = data[row][i - 1];
        }
        data[row][c1] = tmp;
    }

    public void reverseRow(int row, int c1, int c2) {
        String tmp;
        while (c1 < c2) {
```

```
            tmp = data[row][c1];
            data[row][c1] = data[row][c2];
            data[row][c2] = tmp;
            c1++;
            c2--;
        }
    }

    public void circularUpwardShift(int col, int r1, int r2) {
        String tmp = data[r1][col];
        for (int i = r1; i < r2; i++) {
            data[i][col] = data[i + 1][col];
        }
        data[r2][col] = tmp;
    }

    public void circularDownwardShift(int col, int r1, int r2) {
        String tmp = data[r2][col];
        for (int i = r2; i > r1; i--) {
            data[i][col] = data[i - 1][col];
        }
        data[r1][col] = tmp;
    }

    public void reverseColumn(int col, int r1, int r2) {
        String tmp;
        while (r1 < r2) {
            tmp = data[r1][col];
            data[r1][col] = data[r2][col];
            data[r2][col] = tmp;
            r1++;
            r2--;
        }
    }

    public static String[][] additionCode = new String[][]{
            {"01", "10", "11", "00"},
            {"10", "11", "00", "01"},
            {"11", "00", "01", "10"},
            {"00", "01", "10", "11"}
    };
    public static String[][] subsractionCode = new String[][]{
            {"11", "10", "01", "00"},
            {"00", "11", "10", "01"},
            {"01", "00", "11", "10"},
```

```
            {"10", "01", "00", "11"}
    };

    public void performAddition(String[] key) {
        performOperation(key, true);
    }

    public void performSubstraction(String[] key) {
        performOperation(key, false);
    }

    public void performOperation(String[] key, boolean op) {
        int r, c;
        String[][] operationCode = (op) ? additionCode :
subsractionCode;
        for (int i = 0; i < countColumns(); i++) {
            for (int j = 0; j < countRows(); j++) {
                r = Integer.parseInt(data[j][i], 2);
                c = Integer.parseInt(key[j], 2);
                data[j][i] = operationCode[r][c];
            }
        }
    }

    public int[] toArray(String[] key) {
        // return array of key + data;
        int k, keySize = key.length * 2;
        int[] array = new int[countColumns() * countRows() * 2 +
keySize];
        //append the key first
        for (k = 0; k < keySize; k = k + 2) {
            String str = key[k / 2];
            array[k] = Integer.parseInt(str.charAt(0) + "");
            array[k + 1] = Integer.parseInt(str.charAt(1) + "");
        }
        // then add the data
        for (int i = 0; i < countColumns(); i++) {
            for (int j = 0; j < countRows(); j++) {
                array[k] = Integer.parseInt(data[j][i].charAt(0)
+ "");
                array[k + 1] =
Integer.parseInt(data[j][i].charAt(1) + "");
                k = k + 2;
            }
        }
```

```
        return array;
    }

    public List<Point> toPoints() {
        List<Point> list = new ArrayList<>();
        for (int i = 0; i < countColumns(); i++) {
            String pointCode = "";
            for (int j = 0; j < countRows(); j++) {
                pointCode += data[j][i];
            }
            int x = Integer.parseInt(pointCode.substring(0,
pointCode.length() / 2), 2);
            int y =
Integer.parseInt(pointCode.substring(pointCode.length() / 2),
2);
            if (x == 0 && y == 0) {
                list.add(Point.getInfinity());
            } else {
                list.add(new Point(x, y));
            }
        }
        return list;
    }

    @Override
    public String toString() {
        StringBuilder sbResult = new StringBuilder();

        for (int i = 0; i < countRows(); i++) {
            for (int j = 0; j < countColumns(); j++) {
                sbResult.append(data[i][j]);
                sbResult.append("\t");
            }
            sbResult.append("\n");
        }

        return sbResult.toString();
    }

    public void log(String t, int op, int a, int b, int c, int
d) {
        System.out.println("subkey : " + t + "/" + op + "/" + a
+ "/" + b + "/" + c + "/" + d);
        Helpers.saveSubKey(t, op, a, b, c, d);
    }
```

```
}
```

```java
import java.math.BigInteger;

public class Point {

   private BigInteger x;
   private BigInteger y;
   private boolean isInfinity;

   private static Point INFINITY;

   public Point(BigInteger x, BigInteger y) {
       this.x = x;
       this.y = y;
       isInfinity = false;
   }
   public Point(Point p) {
       this.x = p.getX();
       this.y = p.getY();
       isInfinity = p.isInfinity();
   }
   public Point(long x, long y) {
       this( BigInteger.valueOf(x), BigInteger.valueOf(y));
   }
   public Point() {
       this.x = this.y = BigInteger.ZERO;
       isInfinity = true;
   }
   public static Point make(int[] array){
       String x, y;
       x = y = "";
       for (int i = 0; i < 2 * ECC.PAD; i++) {
           if(i<ECC.PAD)
               x += array[i];
           else y += array[i];
       }
       return new Point(Integer.parseInt(x, 2),
Integer.parseInt(y, 2));
   }

   public BigInteger getX() {
       return x;
   }
```

```java
    public BigInteger getY() {
        return y;
    }

    public boolean isInfinity() {
        return isInfinity;
    }

    public Point negate() {
        if (isInfinity()) {
            return getInfinity();
        }
        return new Point(x, y.negate());
    }

    public static Point getInfinity() {
        if(INFINITY == null)
            INFINITY = new Point();
        return INFINITY;
    }

    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (other instanceof Point) {
            if (this.x != null) {
                Point otherPoint = (Point)other;
                // no need to check for null in this case
                return this.x.equals(otherPoint.x) &&
                        this.y.equals(otherPoint.y);
            } else {
                return other == getInfinity();
            }
        }
        return false;
    }

    @Override
    public String toString() {
        if (isInfinity()) {
            return "INFINITY";
        } else {
            return "(" + x.toString() + ", " + y.toString() +
")";
```

```java
        }
    }

    @Override
    public int hashCode(){
        if (this.isInfinity) {
            return x.hashCode() * 31 + x.hashCode();
        }
        return 11;
    }
}
```

```java
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

/**
* The private key of the El Gamal Elliptic Curve Cryptography.
*
* The key consists of: c, the elliptic curve used in the
calculations, k is the
* private key, a randomly-generated integer, satisfying 1 <= k
< p-1.
*/
public class PrivateKey {

    private EllipticCurve c;
    private BigInteger k;

    public PrivateKey(EllipticCurve c, BigInteger k) {
        this.c = c;
        this.k = k;
    }

    public PrivateKey(String pathFile) {
        try {
            List<String> lines =
Files.readAllLines(Paths.get(pathFile),
StandardCharsets.UTF_8);
            BigInteger a = new BigInteger(lines.get(0), 16);
            BigInteger b = new BigInteger(lines.get(1), 16);
            BigInteger p = new BigInteger(lines.get(2), 16);
            BigInteger g1 = new BigInteger(lines.get(3), 16);
```

```
            BigInteger g2 = new BigInteger(lines.get(4), 16);
            BigInteger k = new BigInteger(lines.get(5), 16);
            EllipticCurve eC = new EllipticCurve(a, b, p, new
Point(g1, g2));
            this.c = eC;
            this.k = k;
        } catch (Exception e) {

        }
    }

    public void setCurve(EllipticCurve c) {
        this.c = c;
    }

    public EllipticCurve getCurve() {
        return c;
    }

    public void setKey(BigInteger k) {
        this.k = k;
    }

    public BigInteger getKey() {
        return k;
    }

    public Point getBasePoint() {
        return c.getBasePoint();
    }
}
```

```
import java.io.File;
import java.io.PrintStream;
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

/**
* The key consists of: c, the elliptic curve used in the
calculations, pK, the
* point obtained from k * G, where k is the corresponding
private key and G is
```

```
* the base point of c.
*/
public class PublicKey {

    private EllipticCurve c;
    private Point pK;

    public PublicKey(EllipticCurve c, Point pK) {
        this.c = c;
        this.pK = pK;
    }

    public PublicKey(String pathFile) {
        try {
            List<String> lines =
Files.readAllLines(Paths.get(pathFile),
StandardCharsets.UTF_8);
            BigInteger a = new BigInteger(lines.get(0), 16);
            BigInteger b = new BigInteger(lines.get(1), 16);
            BigInteger p = new BigInteger(lines.get(2), 16);
            BigInteger g1 = new BigInteger(lines.get(3), 16);
            BigInteger g2 = new BigInteger(lines.get(4), 16);
            BigInteger pK1 = new BigInteger(lines.get(5), 16);
            BigInteger pK2 = new BigInteger(lines.get(6), 16);
            EllipticCurve eC = new EllipticCurve(a, b, p, new
Point(g1, g2));
            Point eCP = new Point(pK1, pK2);
            this.c = eC;
            this.pK = eCP;
        } catch (Exception e) {

        }
    }

    public EllipticCurve getCurve() {
        return c;
    }

    public void setCurve(EllipticCurve c) {
        this.c = c;
    }

    public Point getKey() {
        return pK;
    }
```

```
    public void setKey(Point pK) {
        this.pK = pK;
    }

    public Point getBasePoint() {
        return c.getBasePoint();
    }
}
```
(Benaich, 2015)