Elgamal Encryption using Elliptic Curve Cryptography

Rosy Sunuwar, Suraj Ketan Samal
CSCE 877 - Cryptography and Computer Security
University of Nebraska- Lincoln
December 9, 2015

1. Abstract

The future of cryptography is predicted to be based on Elliptic Curve Cryptography(*ECC*) since *RSA* is likely to be unusable in future years with computers getting faster. Increasing *RSA* key length might not help since it would also make the encryption and decryption process slower. A 256-bit *ECC* is considered to be equivalent to 3072-bit *RSA*. Using *ECC* to encrypt data is known to provide the same security as *RSA* but much more efficient in implementation than *RSA*. However, it is slower than symmetric key encryption (e.g *AES*) and hence rarely used for encrypting actual messages. Elgamal encryption using *ECC* can be described as analog of the Elgamal cryptosystem and uses Elliptic Curve arithmetic over a finite field. In this project, we visualize some very important aspects of *ECC* for its use in Cryptography. We explore Elgamal encryption using Elliptic curves and understand its challenges to encrypt data. We also present an approach for fast encryption and compare our results with other popular symmetric and public key cryptosystems. We also implement some basic attack techniques for *ECC* cryptosystems namely, Naive Linear search O) and Baby Step Giant step O).

2. Introduction

Elliptic curve cryptography (*ECC*) is a public-key cryptography system which is based on discrete logarithms structure of elliptic curves over finite fields. *ECC* is known for smaller key sizes, faster encryption, better security and more efficient implementations for the same security level as compared to other public cryptography systems (like *RSA*). *ECC* can be used for encryption (e.g *Elgamal*), secure key exchange (*ECC Diffie-Hellman*) and also for authentication and verification of digital signatures.

The security of *ECC* is based on a trapdoor function where it is assumed that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible. This is called **E**lliptic **C**urve **D**iscrete **L**ogarithm **P**roblem(*ECDLP*) which is considered to be computationally infeasible to solve. It is explained in further details below.

ECC is similar to *RSA* in application. The modular multiplication and modular exponentiation in *RSA* is equivalent to *ECC* operations of addition of points on an elliptic curve and multiplication of a point on an elliptic curve by an integer respectively. In *RSA*, the security is based on the assumption that it is difficult to factor a large integer composed of two large prime factors. So, *RSA* needs a large key size to be secure and unbreakable. But for *ECC,* it is possible to use smaller primes, or smaller finite fields, with elliptic curves to achieve same

degree of security. ECC is successfully being used in variety of areas like BitCoin currency, OpenSSH (v5.7 and above) key exchange and TLS(RFC 4492) certificates.

Several discrete logarithm-based protocols have been modified to use elliptic curves. The most common version of *ECC* is *ECC* with Diffie Hellman which is same as Diffie Hellman but it uses elliptic curve math for secure key exchange. Other examples are Elliptic Curve Digital Signature Algorithm(*ECDSA*), Edwards-curve Digital Signature Algorithm(*ECDSA*) and *ECMQV* Key agreement scheme.

The organization of this report is as per below. In Section 3, we discuss basic theory behind Elliptic curves, its operations over finite field, the hardness of Elliptic Curve Discrete Logarithm(*ECDLP*) problem and Elgamal encryption/decryption using ECC. Section 4 describes a visualization of Elliptic Curves(*EC*) over finite field and its operations using JavaPlot library[5]. Section 5 covers our implementation of ECC Elgamal encryption using the JECC library[7]. Section 6 presents the results of ECC encryption/decryption and compares them with DES, AES, RSA and ECIES algorithms and describes an approach using *k-Table* to improve the encryption efficiency. Finally in Section 7, we discuss some basic attack approaches for ECC Cryptosystems.

3. Background and Theory

3.1 Elliptic Curves over Finite Field

ECC uses an elliptic curve over a finite field (p) of the form:-

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

The curve defines a finite field consisting of points that satisfy this equation along with infinity(∞) as the identity element. The value of a and b determines the shape of the curve. Only those curves which doesn't have repeated factors for $x^3 + ax + b$ are used in cryptography. One can check that by calculating $4a^3 + 27b^2 \quad 0 \pmod p$. Here, modulo prime p is used to fix the range of the curve. The order(n) of the curve is the total number of points that lie on the curve including the point at infinity.

Some examples of elliptic curves are given in the figure below:-

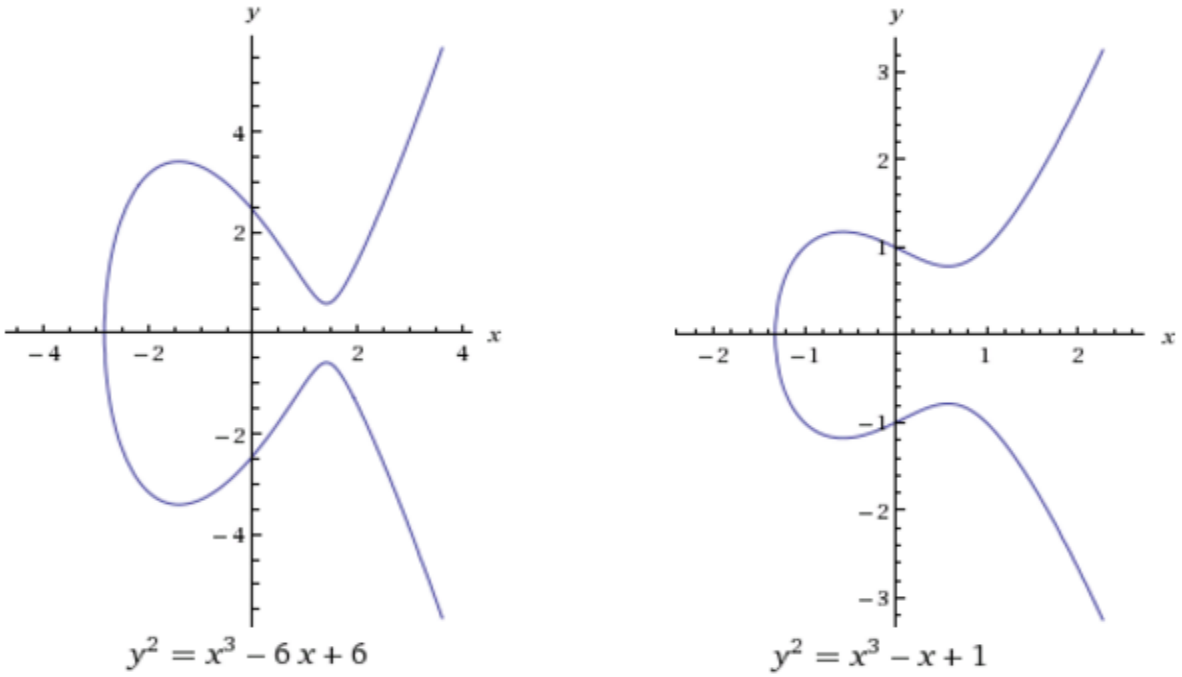

Figure 1: Elliptic Curves

Elliptic curves posses some great properties for use in Cryptography. The arithmetic operations used in elliptic curves are different from the standard algebraic operations. To add two distinct points P and Q in the curve, a line is drawn through them. This line will intersect the curve at a third point, -R. Then -R is reflected in the x-axis to get the point R. This point is the result of addition of P and Q. i.e. P + Q = R. If the point P and Q are vertical i.e. Q=(-P), then the line will not intersect the elliptic curve at a third point. In such case, P + (-P) = O (infinity).

To add a point P to itself, a tangent line to the curve is drawn at the point P. If the point doesn't lie on the x-axis, then this tangent intersects the elliptic curve at one other point, -R. Then -R is reflected in the x-axis to get the result R i.e. P + P = 2P = R. This operation is also referred to as point doubling. It is a common way to achieve multiplication of point in elliptic curves.

Algebraically, the addition of points P (x_1, y_1) and Q (x_2, y_2) in ECC can be expressed as:-

P + Q = R and coordinates of R (x_3, y_3) are given by

$x_3 = m^2 - x_1 - x_2 \bmod p$ and $y_3 = -y_1 + s(x_1 - x_3) \bmod p$

And m is the slope of line through P and Q calculated as $m = (y_1 - y_2) / (x_1 - x_2) \bmod p$.

If P = Q, then P + P = 2P = R and coordinates of R (x_3, y_3) are given by

$x_3 = m^2 - 2x_1 \bmod p$ and $y_3 = -y_1 + s(x_1 - x_3) \bmod p$

And m is the slope of the line given as $m = (3x_1^2 + a) / (2y_1)$

In practice, ECC allows use of whole number points only and within a fixed range. That is, the points are rolled over by using modular operation with respect to a prime to confine points within a range, similar to RSA. As a result, the curves used for cryptography doesn't look straightforward as in the figure above. It consists of the curve wrapped around at the edges and only the positive whole number points in the curve are included. The figure 2 below shows points on one such ECC curve. For an elliptic curve modulo prime to be used for cryptographic purposes, it's order n (i.e. the number of points), should be of comparable size compared to prime.

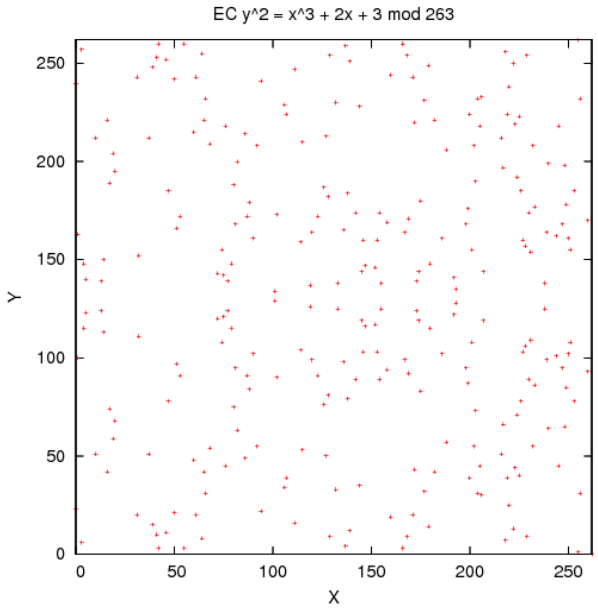

Figure 2: Points on ECC curve over a finite field[9]

3.2 Elliptic Curve Discrete Logarithm Problem (ECDLP)

The basic Discrete Logarithm Problem requires to find *k* where $x^k = y$ and x,y belong to the same group G. The elliptic curve version requires to find *k* where P **o** k = Q and points P, Q belong to a set of points G on an elliptic curve. This problem is known to be computationally difficult and hence can be used to achieve a higher level of security in cryptosystems. All currently known algorithms to solve the problem are exponential. A 256-bit *ECC* is considered to be equivalent to 3072-bit RSA.

3.3 Elgamal Encryption using Elliptic Curve Cryptography(E^3C^2) [1]

Elgamal cryptosystem is directly based on ECDLP described above. Elgamal Cryptosystem was first described by Taher Elgamal in 1985. The concept lies in hiding the message *m* using α^k and β^k where α is a primitive root of a large prime *p* and *k* a random

[1] We abbreviated the Elgamal encryption using ECC as *EEECC* or $\boldsymbol{E^3C^2}$ for simplicity. Note, it resembles the order of *x* and *y* powers on the curve (i.e $\boldsymbol{x^3}$ and $\boldsymbol{y^2}$).

integer. Note, $\beta = \alpha^a$ where *a* is a secret known only to the receiver. The following example describes the process in greater detail:

Let's say *Abbie(A)* wants to send a secure message *m* to *Brooke(B)* over a channel that is also accessible by *Ceaser*(C). *Brooke* chooses a secret *a* , a large prime *p* and a primitive root α. *Brooke* also computes $\beta = \alpha^a$ and makes (α, β, p) public. While sending a message, *Abbie* uses a random *k* and computes (α^k, $\beta^k m$) and sends to *Brooke*. Note, *Ceaser* cannot retrieve the original message as he will then need to solve the ECDLP problem. However, *Brooke* using her secret *a* can retrieve the message *m* using

$$(\alpha^k)^{-a} * (\beta^k m) = (\alpha^a)^{-k} * (\beta^k m) = (\beta^{-k}) * (\beta^k m) = m$$

The Elliptic curve version of the encryption is the analog of Elgamal encryption where α and β are points on the Elliptic curve and multiplication operations replaced by addition and exponentiation replaced by multiplication (using ECC arithmetic). We describe the process below.

Let's assume *Abbie*(A) wants to send a secure message *m* to *Brooke*(B) using a communication channel that can be accessed by *Ceaser*(C). *Brooke* chooses an elliptic curve *C* (of form y^2 = x^3 + bx + c) modulo *p* and secret integer *a*. A point α (x1, y1) is chosen to lie on the curve and the point β (x2, y2) is calculated using ECC arithmetic using

$\beta = a \circ \alpha$ (adding α a times)

The values (p, α, β) and the curve C are made public. Note, *Ceaser*(C) will not be able to decipher the secret *a* as it will involve solving the ECDLP problem. Now *Abbie*(A) breaks the message *m* is into smaller blocks(if required) and encodes each block as an integer modulo the prime number *p*. *Abbie*(A) then uses a random integer *k* and computes two additional points on the curve namely,

$r\ (x3, y3) = (x3, k \circ \alpha)$ and $t\ (x4, y4) = (x4, m \dot{+} k \circ \beta)$

where ($\circ$) represents multiplication and ($\dot{+}$) represents addition in ECC arithmetic. The pair *(r,t)* is sent to to *Brooke*. Note, the message is embedded into the *y*- co-ordinates of the points and hence it is sufficient to send *(y3, y4)* to Brooke. *Brooke(B)* now uses the following method to calculate *m*:

$y4 - (a \circ y3) = (m \dot{+} k \circ \beta) - a \circ (k \circ \alpha) = m \dot{+} a \circ k \circ \alpha - a \circ k \circ \alpha = m$

where (-) represents negative addition using ECC arithmetic.

Note, Ceaser(C) can intercept the values ($k.\alpha$, $m + k.\beta$) , but because the discrete logarithm is difficult to compute, will not be able to obtain *k* or *m*. For a practical example using integers, readers are encouraged to see Trappe-Washington[8].

4. Visualization of ECC and ECC over finite field

We used *JavaPlot* [5] library for visualizing basic mathematical operations in Elliptic Curve. The library enables us to visualize different mathematical functions using basic charts by specifying a formula for the graph. We took advantage of this feature to draw our ellipse but had to modify considerable part of the library source code to fit our visualization needs. Our tool helps visualize adding of two points and doubling a point on the curve. We took our curve to be

$y^2 = x^3 - x + 1$ and fixed it for our visualization purposes. We then implemented the addition of two points and doubling a point. The tool recognizes if same points are given to add as input, and doubles that point. For these mathematical operations, the two points should lie on the ellipse, else it will fail. So we took a number of points on the ellipse and allowed the user to choose from these points. On clicking the "Add two points" button, the visualization starts. The resulting point is also displayed on the bottom of the screen.
A screenshot of one such visualization is given below. It shows the addition of two points, P(-1,-1) and Q(0,1). The result is point R(5, -11).

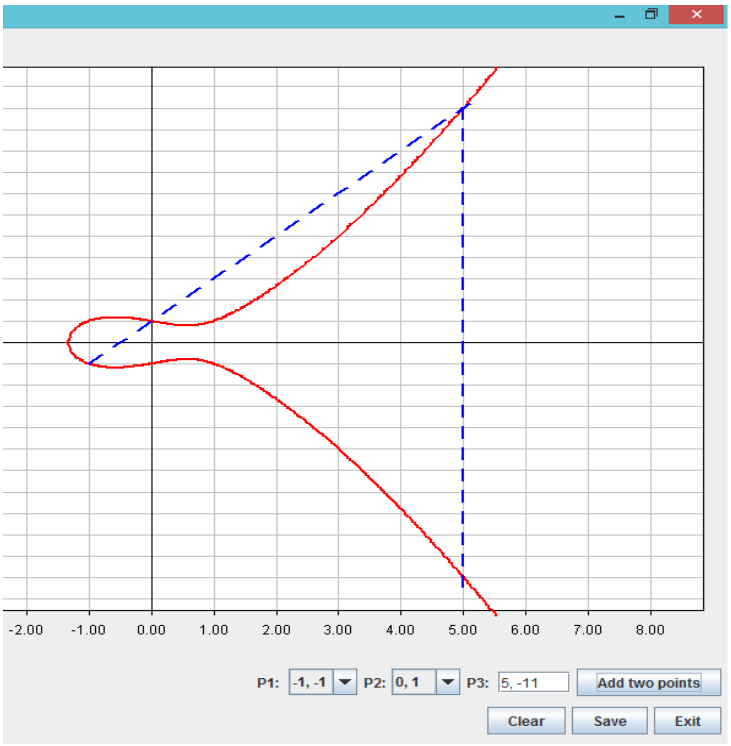

Figure 3: Visualization of ECC addition using our visualization tool.

Note that in ECC, *multiplication* is achieved by repeated *addition*. To get *xα*, repeated addition of *α* is used as *α + α = 2α, α+2α=3α* and so on until we achieve the required value. For visualization purposes, we only implemented doubling process which is multiplying by 2 i.e. *2α.*

5. Implementation of Elgamal ECC Cryptosystem (E^3C^2)

We used JECC[7] library to implement the Elgamal encryption using Elliptic curves. JECC is an open-source java based library that provides APIs for generating keys using existing ECC curves, creating new ECC curves, doing ECC arithmetic (point addition and multiplication) and encrypting and decrypting integers using the public/private keys.

For our implementation, we choose a fixed curve *C*, fixed prime modulo *p* and a fixed point *P(x,y)* for α provided by the library. The various curve parameters and *P(x,y)* chosen are described below:

Curve Parameters	Value
Prime *p* (256 bits)	FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
a	FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
b	5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B
α = P(x,y)	*x* = 6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296 *y* = 4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5
n (order of the curve)	FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551

Table 1 : Various ECC parameters used by our cryptosystem. [2]

5.1 Encryption and Decryption

Encryption:

A message *m* of arbitrary length is divided into blocks of *20* bytes(m*0*, m*1*…...m*n*). We assume the message to be *ascii* characters and hence each character of message can be encoded using a 8-bit integer(0,255) at the source and similarly decoded at the receiver. The encoded integer is then embedded into the *x*- coordinate and corresponding *y* coordinate is found such that *(x,y)* lies on the selected elliptic curve. Choosing a random k*i,* r*i* = (k*i* ₒ *α)* and ht = *H (*k*i* ₒ *β)* are calculated where *H* is the *SHA-1* hash function. We also calculate *htmi* = *ht* ⊕ m*i* (⊕is bitwise XOR). Note the *SHA-1* output is 20 bytes which encrypts a block of message(20 bytes). The encrypted block e*i* is then obtained by appending r*i* to *ht*m*i* to get

$$e_i = r_i \mid (ht m_i)$$

The final encrypted message is obtained by combining *E* = (e*0*e*1*e*2*e*3*…..e*n)*.

Decryption:

The encrypted message *E* is divided into chunks (e*0*e*1*e*2*e*3*…..e*n*), and for each block e*i,* r*i* and *htmi* are calculated. Note, it is easy to decompose as the size of hash output is known to the reciever. A message block m*i* is calculated as follows:

[2] This is the same curve as CloudFlare's *ECC* curve for *ECDHE* and also the one used by *Google.com* [10].

$H(a \circ ri) \oplus htmi = H(a \circ ki \circ \alpha) \oplus (ht \oplus mi) = H(ki \circ \beta) \oplus H(ki \circ \beta) \oplus mi = mi$

The entire message is finally obtained by combining M = (m*0*m*1*m*2*...m*n*).

Fast Multiplication:

JECC uses a technique to compute fast multiplication of points on the elliptic curve E. For multiplying a scalar *s* (of *t* bits) to a point *P*, first a table is created to store points generated by recursively doubling the point *P* to obtain *2P, 4P,* 2^tP. This requires *t* additions. Now *s* can be represented in binary (b*t*.....b*2*b*1*b*0*) and hence $s = 2^t + 2^{t-x1} + \ldots + 2^{t-xn.}$ (for every non-zero bit b*i*). Then *sP* can be calculated by performing additional *t* additions (at max) by using the corresponding values of 2^xP from the existing table. This reduces the number of additions to (*2t = log(s)*) from *s* additions and hence considerably faster than naive approach.

Calculating square root in terms of modular multiplication:

For any point *P(x,y)* calculating *y* using square root of ($x^3 + ax + b$) mod *p* can be expensive, however using Euler's theorem this can be efficiently implemented as follows:

From Euler's theorem, we have

$a^{(p-1)} = 1 \pmod p$ since *p* is prime.

$a^{p+1} = a^2 \pmod p$ or $a^{(p+1)/2} = a^{1/2} \pmod p$

So instead of computing $(x^3 + ax + b)^{1/2} \bmod p$, we can compute $(x^3 + ax + b)^{(p+1)/2} \pmod p$ which is relatively cheaper.

Customization:

We customized the existing *JECC* implementation slightly to be able to encrypt a file containing ascii characters for efficiency. We treated each line of the file as a separate block and padded it with spaces till the size became a multiple of 20 bytes. The whole line was then encrypted using a random *k* for the whole block (note the block is a multiple of 20). The encrypted text was output as a separate line. For each line *i* (m0...mt), the encrypted text is calculated as

$ei = ri \mid htm0 \mid htm1 \mid htm2\ldots.\mid htmt$

The decryption was handled accordingly by using the same *ri* for the entire line.

We also implemented other encryption algorithms, namely AES, DES, RSA (using by Java Crypto library) and ECIES(using a third-party library [12]) to compare our results with them.

6. Encryption/Decryption Results and Comparison with other approaches

6.1 Encryption and Decryption with ECC on varying input sizes

We tested our ECC implementation by encrypting ascii text files of various sizes (multiples of *119 KB*) and measured the time taken for encryption. The output encrypted text size turned out to be about *3.9* times the original text. Decryption was observed to be about *7* times faster than encryption. The obvious reason is because decryption involves a single multiplication compared to two multiplications required during the encryption process.

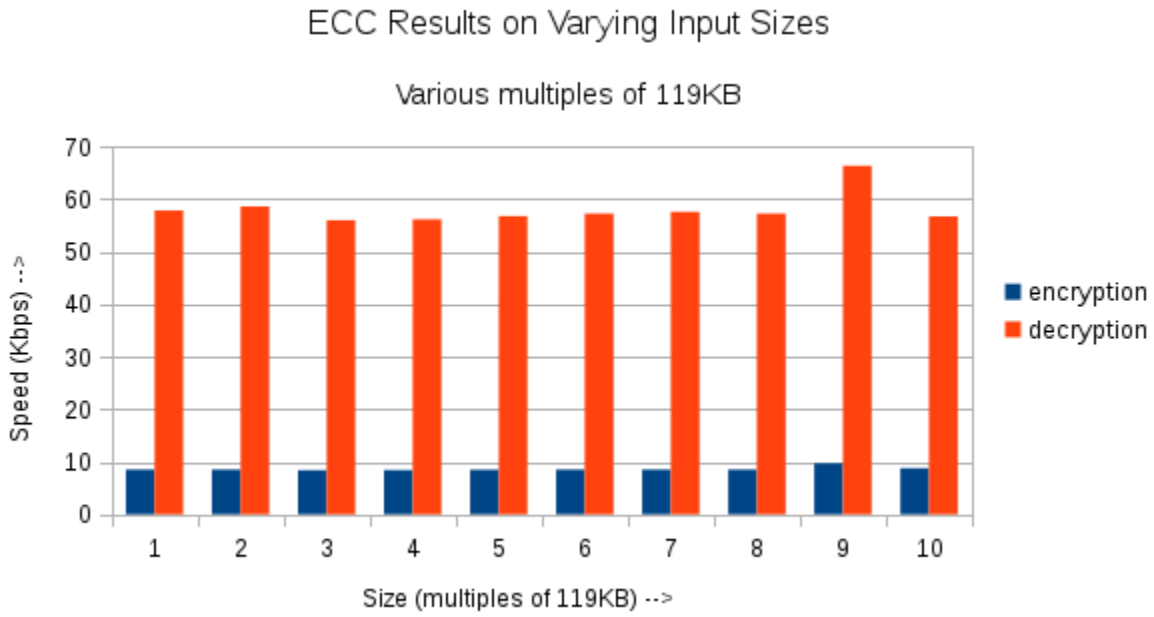

Figure 4: Encryption vs Decryption on various input sizes using E^3C^2.

6.2 Comparison with AES, DES, RSA and ECIES

We compared our E^3C^2 implementation with other existing algorithms (AES, DES, RSA and ECIES) using the various parameters as described in *Table 2* below.

Algorithm	Properties	Key Size
AES	ECB/PKCS5 Padding	128 bit
DES	ECB/PKCS5 Padding	56 bit
RSA	ECB/PKCS1 Padding	1024 bit
ECIES	CBC/AES128_HMAC_SHA1	256 bit
E^3C^2(Our	ECC_SHA1	256 bit

implementation)		

Table 2 : Parameters used by other algorithms for comparison.[3]

6.2.1 Encryption Speed

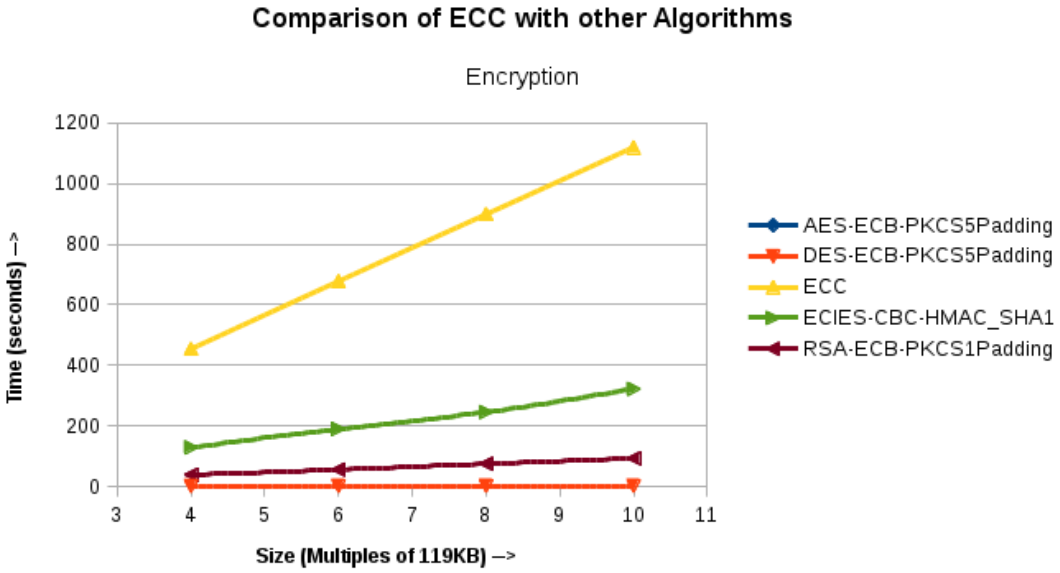

Figure 5: Comparison of Encryption times of various algorithms on various input sizes.

Encryption speed is an important deciding factor for choice of an algorithm for encrypting data, the reason why *AES* is so popular. As we expected, *DES* turned out to be fastest because of its low key size but it is highly insecure. Among the secure algorithms, *AES* came up to be fastest as expected (the difference was in the order of milliseconds with *DES* and hence they pretty much overlap in Figure 5). Our E^3C^2 implementation turned out to be worst, even worser than *RSA* and *ECIES*. *ECIES* turns out to be better than E^3C^2 because it uses *ECC Diffie-Hellman* to exchange the secret key and then uses *AES* to encrypt the actual data.

6.2.2 Cipher Text to Plain Text Ratio:

[3] The key sizes of the algorithms were chosen such that they provide at most the same level of security as *ECC*(except for DES).

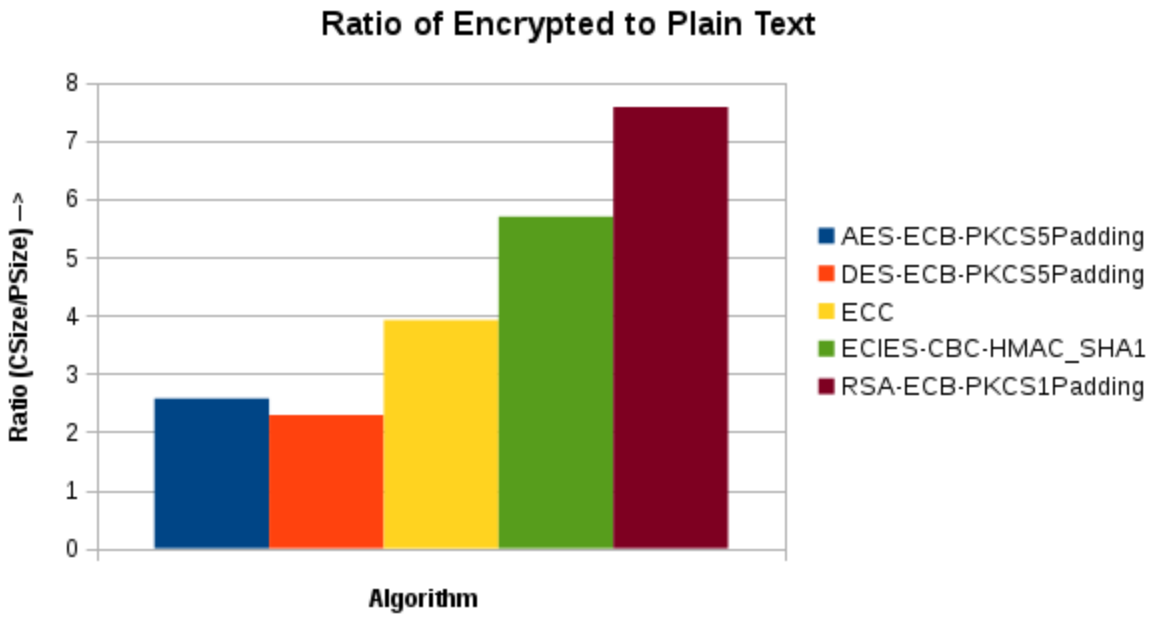

Figure 6: Ratio of encrypted to plain text for various algorithms.

Interestingly, it's the cipher text that gets sent over the communication channel, hence it is important that the ratio of the *cipher-text* to *plain-text* is not too large. Figure-5 shows the cipher-text to plain-text ratio for various algorithms and its interesting to note that E^3C^2 does better than the *public-key* algorithms but ends up being more than double the size in comparison to the standard *AES*. (*5.8* compared to *2.3* for *AES*). This isn't a good result in terms of using E^3C^2 for encrypting and decrypting data.

6.2.1 Decryption Speed:

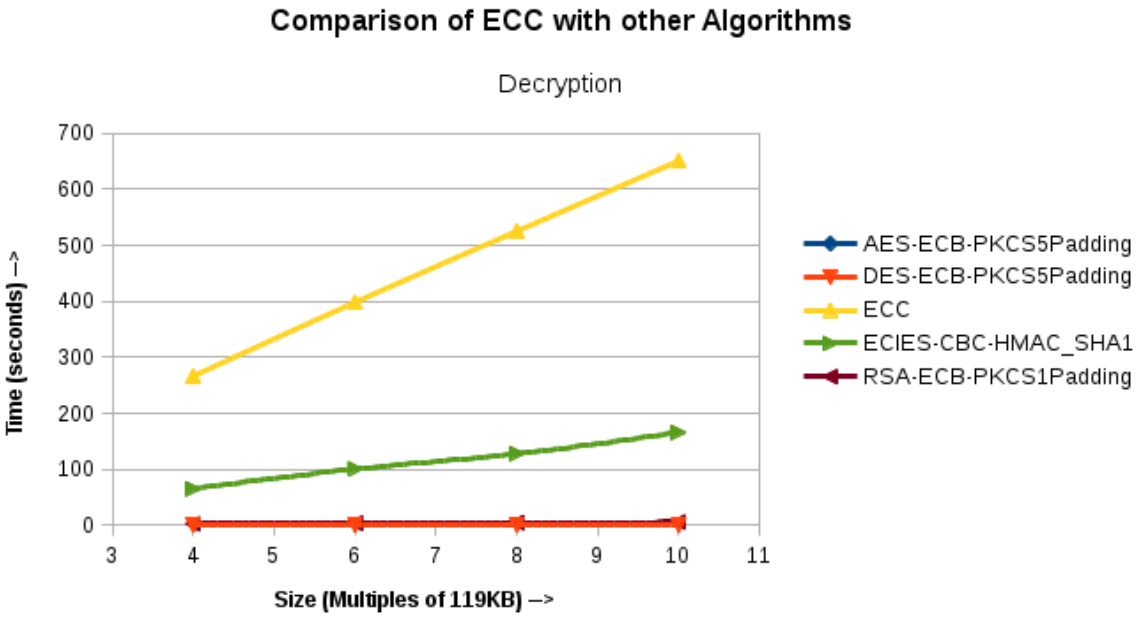

Figure 7: Comparison of Decryption times of various algorithms on various input sizes.

The observed behaviour is similar to as of *encryption*. We observe that decryption is faster as compared to encryption in *AES* (as expected based on the theory behind *AES*). Interestingly, *RSA* decryption is observed to be comparable with the classic symmetric key algorithms (*AES, DES*). E^3C^2 decryption is still the slowest, but the decryption times are much better than the encryption times.

6.2.2 Detailed Analysis of E^3C^2 encryption:

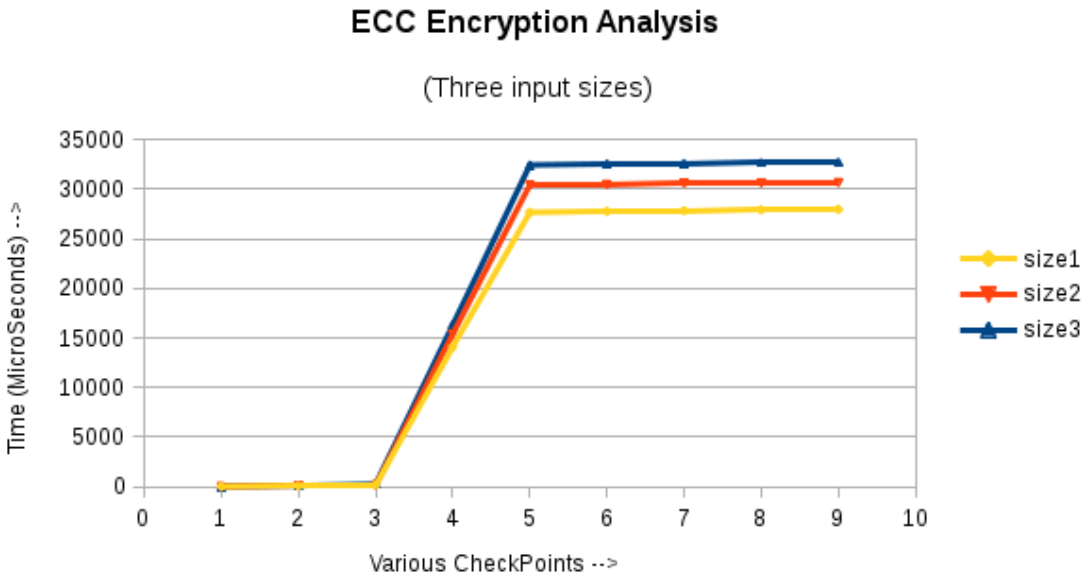

Figure 8: Timestamps of various checkpoints in E^3C^2 encryption

On detailed analysis by recording timestamps at various stages of the encryption process, we observed that the *multiplication* operations are the most expensive .The intervals [*3-4*] and [*4-5*] shown in *Figure 8* represent these operations and take about ~90% of the entire encryption time.

6.2.2 Detailed Analysis of E^3C^2 Decryption:

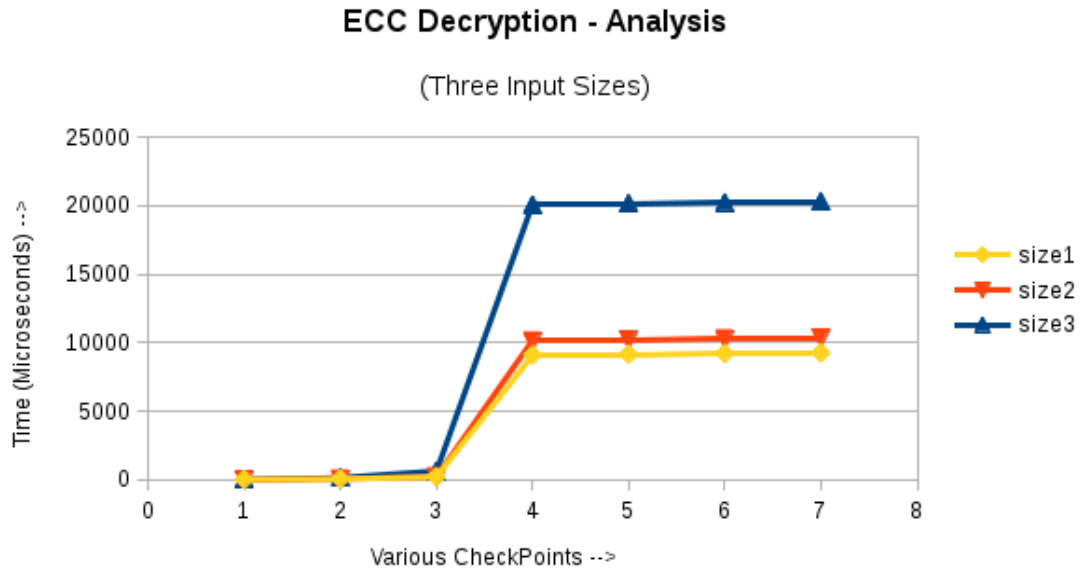

Figure 9: Timestamps of various checkpoints in E^3C^2 decryption

Similar observation was seen with decryption.The intervals [*3-4*] shown in *Figure 8* represent operation that contribute to most of the decryption time and this turned out to be the *multiplication* operation.

6.3 A novel approach for fast ECC encryption using k-Table (E^3C^2K)

As mentioned earlier, we observed that the two multiplication operations (using ECC arithmetic) consume most of the time (>90%), shown in Figure 8. The multiplications are $k \circ \alpha$ and $k \circ \beta$ where α and β are fixed for the entire encryption process and *k* is chosen at random. Rather than choosing *k* and doing the multiplication during encryption, our approach involves pre-computing a *k*-table for various random values of *k*. For this, we define a *k*-Tuple as $(k, k \circ \alpha, k \circ \beta)$ and pre-compute many such tuples for various random values of *k* and store it in an array which we call as *k*-table. Pre-computing such multiplications is much faster due to the ability to cache various existing multiples of α and β using fast-multiplication explained above. So, instead of performing the expensive multiplication operations during encryption, we reduce the actual encryption time by a considerable amount. The results look promising as now overall encryption time is comparable to other symmetric and public-key encryption algorithms (*Figure 11 and Figure 12*).

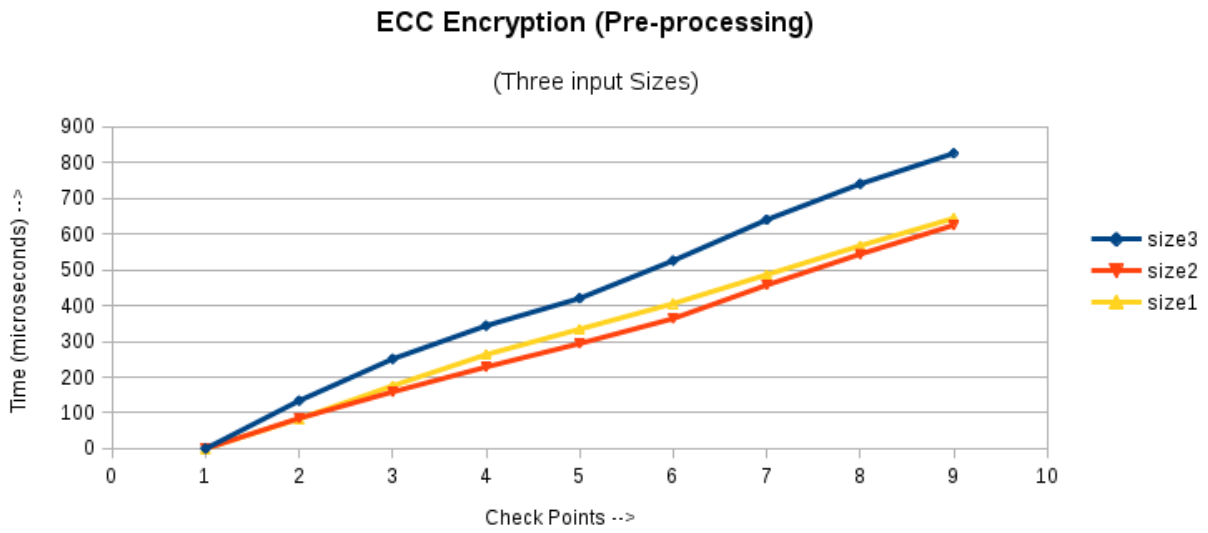

Figure 10: Timestamps of various checkpoints in E^3C^2K encryption

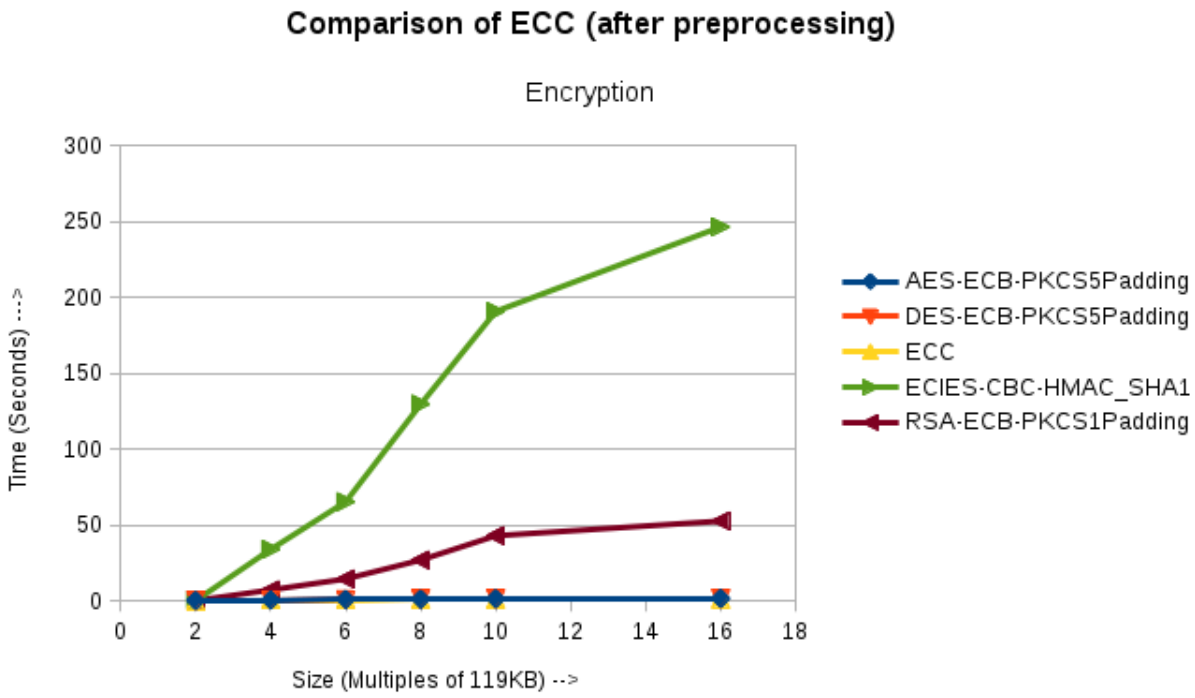

Figure 11: Comparison of E^3C^2K encryption with other algorithms.

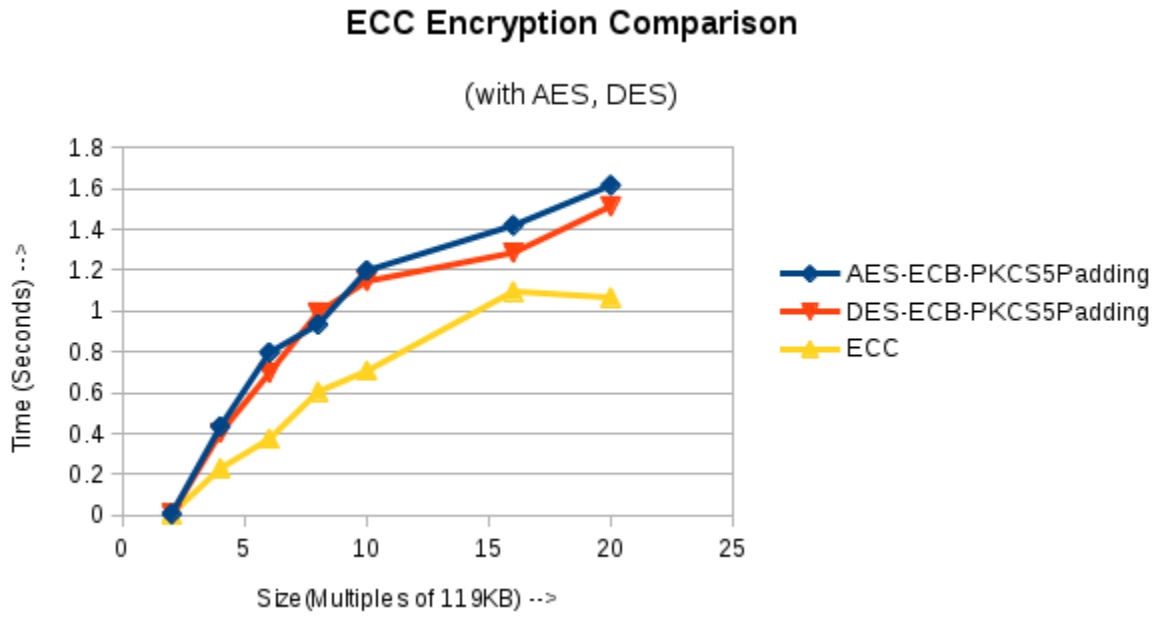

Figure 12: E^3C^2K encryption - A closer look with *AES* and *DES*

7. Attacks on ECC

The security of ECC is based on Elliptic Curve Discrete Logarithm Problem (ECDLP). That is, given α and β, it is easy to compute $\beta = \alpha^k$ but it is difficult to compute $k = \log_\alpha \beta$. In ECC, α^k is calculated as k*α by using multiple additions of α on the curve. The ECC is considered broken if there we are able to calculate k. But for large enough values of α and β, it is considered to take impossible time to do a brute force attack.

We explored some algorithms for attacking this Discrete Logarithm Problem.

7.1 Linear Search

One naive approach of attacking DLP is Linear Search, which consists of exhaustively searching through the values of k to find β. i.e. computing α, 2α, 3α,.....,xα and checking whether they are equivalent to β or not. It's complexity is O(n) and is not a very good algorithm for large key sizes. It is infeasible in practice.

7.2 Baby-steps giant-steps

This algorithm is a modification of the naive approach but improves upon the time complexity of the search to O). It is based on the fact that any integer can be represented as a sum of three arbitrary integers as:-

Then based on this, β can be rewritten as:-

where ¯, and

The multiple of α is calculated in two steps as jα in small steps and imα in larger steps, and hence the name Baby-steps giant-steps algorithm. jα is precomputed and stored in memory. Then - imα is computed for each value of i and checked against jα. When they are equal, we can calculate the corresponding value of *k* as $k = im + j$. Thus this algorithm acts as a meet-in-the middle attack. The time complexity is still exponential but is much better than the naive brute force attack.

We implemented this algorithm for breaking our ECC system. Given the public key, we followed the above method to calculate the value of private key k. To check whether the algorithm was working successfully or not, we supplied an encrypted file for the given private-public key pair. And tried to decrypt this file using the calculated private key k. We successfully managed to do so for smaller key sizes up to 16 bits. For some cases, multiple value for k was possible. In such cases, we used all values of k for decryption process and checked which decrypted file actually made sense. For larger key sizes, 32 bits or more, our local system wasn't powerful enough and we got out-of-memory error. In conclusion, the baby-steps giant-steps algorithm trades off time with space complexity, because of its huge memory requirements.

7.3 Pollard's ρ

The Pollard's rho algorithm is another algorithm for computing discrete logarithms. It has same time complexity of O) as Baby-steps giant-steps algorithm but it doesn't have huge memory requirements.

For any given value of α and β, this algorithm tries to find four integers a, b, A and B such that

Once the four integers are found, then the value of k in $\beta = k\alpha$ can be calculated as:-

Since the calculation is carried out in modulo p, the value of k can be computed as:-

$k = (a-A)(B-b)^{-1} \mod p$

The rationale behind this algorithm is that both α and β are elements of same cyclic subgroup. β is calculated as some multiple of α in modulo p. So when a sequence of pair of integers a and b is used to generate a sequence of points $a\alpha + b\beta$, these points are also cyclic. So for distinct pairs (a,b) and (A,B), the calculated points converge to the the same point i.e. $a\alpha + b\beta = A\alpha + B\beta$.

8. Conclusion and Future Work

Many cryptosystems depend upon the assumption that the mathematical problems underlying them are extremely hard to solve. The strength of public key cryptography systems like *RSA* heavily rely on such an assumption. But with the increasing computing ability, these mathematical problems are becoming easier and easier to solve. As a result, many current cryptographic algorithms are feared to become insecure faster than ever. In this scenario, *ECC* appears to be a reasonable alternative. In our view *public-key* cryptosystems are much better and secure as there is no need to exchange a secret key before starting communication. *ECIES* overcomes this drawback by sending the symmetric key using *ECC* and then encrypting the data using symmetric key but it makes the cryptosystem a little complex. However, *ECC* has its own set of problems. The major one being how to choose good curves with efficient arithmetic. Some of these efficient elliptic curves are patented for a range of uses. Therefore *ECC* is still not widely used by the majority of industry. But *ECC* is supported by all modern browsers and most certification authorities offer elliptic curve certificates.[11]. *ECC* computations are known to be very expensive but we have successfully demonstrated an approach to prove that there is considerable scope to make ECC encryption/decryption faster and more practical.

We fixed the elliptic curve for our current visualization implementation. In future, the program can be made to take any ellipse as an input and visualize different elliptic curve operations. Also the program only allows a handful of points on the curve. It can definitely be expanded to include all the points on the ellipse. We could not implement and visualize all the points on Elliptic Curve over the finite field due to time constraints. For more clarity of the basic mathematical operations in ECC, the normal elliptic curve and the curve over finite field can be referenced side by side. We also plan to study the ECC *decryption* process in a little more detail with a hope to find a way to make it faster as well so that it is comparable with other algorithms.

10. References

[1] Koblitz, Neal. "Elliptic curve cryptosystems." *Mathematics of computation* 48.177 (1987): 203-209.
[2] Bos, Joppe W., et al. "Elliptic curve cryptography in practice." *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2014. 157-175.
[3] Wikipedia description of ECC : https://en.wikipedia.org/wiki/Elliptic_curve_cryptography
[4] ECC Introduction: https://www.certicom.com/10-introduction
[5] Trappe and Washington 2nd edition, pages 363-364.
[6] Ceticom ECC Challenge : https://www.certicom.com/index.php/the-certicom-ecc-challenge
[7] Java Elliptic Curve Cryptography(JECC) software : http://jecc.sourceforge.net/
[8] JavaPlot Library:- http://vase.essex.ac.uk/software/JavaPlot/
[9] ECC Points over a finite field: http://www.johannes-bauer.com/compsci/ecc/
[10] An example of a widely used ECC curve: https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/
[11] Attacks on ECC:
http://andrea.corbellini.name/2015/06/08/elliptic-curve-cryptography-breaking-security-and-a-comparison-with-rsa/
[12] Flexiprovider library for ECIES: https://www.flexiprovider.de/overview.html