ELLIPTIC CURVE CRYPTOGRAPHY

By

***SOUMYADIP PAUL* (Roll No:-042844)**
***SAMARJIT MALLICK*(Roll No:-042836)**
***SUTANU SUR*(Roll No:-042832)**

B.E. in INFORMATION TECHNOLOGY

JADAVPUR UNIVERSITY

Under The Guidance Of

Mr. UTPAL RAY

ACKNOELEDGEMENT

We would like to thank a lot of people without whose help and encouragement this project paper would not have been completed.

First and foremost, We thank our guide Mr. Utpal Ray. His suggestions and guidance helped me immensely in understanding the subject.

We would also like to thank our friends for the constant support and help they have provided me all the time.

SOUMYADIP PAUL
SAMARJIT MALLICK
SUTANU SUR

B.E in INFORMATION TECHNOLOGY

JADAVPUR UNIVERSITY

ABSTRACT

The use of Java in developing commercial Internet applications isgrowing very rapidly. A major requirement for e-commerce applications is the provision of security. In this work we consider Elliptic Curve Cryptography (ECC) because of the high level of security it provides with small key sizes. ECC is ideal for use on constrained environments such as pagers, personal digital assistants, cellular phones and smart cards. This project paper gives an overview of Elliptic Curve Cryptography . It includes what is cryptography ? Its classification-Symmetric key (includes block cipher and Stream cipher) & Asymmetric(Public key) key Cryptography . It explains Elliptic Curve groups and arithmetic over Real number Field , Prime number Field & Binary Field with example. It also includes key exchange mechanism , encryption decryption mechanism and why this cryptography is better than the mostly used public key cryptography RSA .

Contents

1 Introduction

1.1 Background

Since the invention of public-key cryptography in 1976 by Whitfield Diffie and Martin Hellman, numerous public-key cryptographic systems have been proposed. All of these systems rely on the difficulty of a mathematical problem for their security.

Over the years, many of the proposed public-key cryptographic systems have been broken, and many others have

been demonstrated to be impractical. Today, only three types of systems should be considered both secure and efficient. Examples of such systems, classified according to the mathematical problem on which they are based, are:

1. Integer factorization problem (IFP).
2. Discrete logarithm problem (DLP)
3. Elliptic curve discrete logarithm problem (ECDLP).

1.2 Elliptic curve cryptosystems

Elliptic curve cryptography (ECC) was proposed by Victor Miller and Neal Koblitz in the mid 1980s.

An elliptic curve is the set of solutions (x,y) to an equation of the form $y^2 = x^3 + ax + b$, together with an extra point O which is called the point at infinity. For applications to cryptography we consider finite fields of p elements, which I will write as $\mathbb{F}_p$ or GF(p). When p is a prime one can think of $\mathbb{F}_p$ as the integers modulo p.

We usually write E for the equation $y^2 = x^3 + ax + b$ and use the notation E($\mathbb{F}_p$) for the set of points (x,y) with coordinates in the field $\mathbb{F}_p$ together with the point O (which is defined over every field).

The set of points on an elliptic curve forms a group under a certain addition rule, which we write using the notation +. The point O is the identity element of the group. When we work over a finite field then this group is necessarily finite (as there are only finitely many points).

Given a point P=(x,y) and a positive integer n we define [n]P = P + P + ... + P (n times). The order of a point P=(x,y) is the smallest positive integer n such that [n]P = O.

We denote < P > by the group generated by P. In other words < P > = {O, P, P+P, P+P+P, . . .}.

***Encryption and Decryption*:-**

The plaintext message m is first encoded to be sent as x-y point P_m. The point P_m will be encrypted as a cipher text and subsequently decrypted.

For the key exchange system, an encryption/decryption system requires a point G on elliptic curve whose order is large value n and an elliptic group F_p (a, b) with parameters a, b and q which is prime Each user A selects a private key n_A and generates a public key $P_A = n_A * G$.

To encrypt and send a message P_m to B, A chooses a random positive integer k and producers the cipher text C_m consisting of the pair of points.

$$C_m = \{k\,G, P_m + k\,P_B\}.$$

A has used B's public key P_B. To decrypt the cipher text, B multiplies the first point in the pair by B's secret key and subtracts the result from the second.

$$P_m + k\,P_B - n_B\,(k\,G) = P_m + k\,(n_B\,G) - n_B\,(k\,G) = P_m.$$

The primary reason for the attractiveness of ECC over systems such as RSA and DSA is that the best algorithm known for solving the underlying mathematical problem (namely, the ECDLP) takes *fully exponential* time. In contrast, *subexponential-time* algorithms are known for underlying mathematical problems on which RSA and DSA are based, namely the integer factorization (IFP) and the discrete logarithm (DLP) problems. This means that the algorithms for solving the ECDLP become infeasible much more rapidly as the problem size increases than those algorithms for the IFP and DLP. For this reason, ECC offers security equivalent to RSA and DSA while using far smaller key sizes.

The attractiveness of ECC will increase relative to other public-key cryptosystems as computing power improvements force a general increase in the key size. The benefits of this higher-strength per-bit include:

1. higher speeds,
2. lower power consumption,
3. bandwidth savings,
4. storage efficiencies, and
5. smaller certificates.

These advantages are particularly beneficial in applications where bandwidth, processing capacity, power availability, or storage are constrained. Such applications include:

1. chip cards,
2. electronic commerce,
3. web servers,
4. cellular telephones, and
5. pagers.

1.3 Why have a challenge ?

The objectives of this ECC challenge are the following:

1. To increase the cryptographic community's understanding and appreciation of the difficulty of the ECDLP.
2. To confirm comparisons of the security levels of systems such as ECC, RSA and DSA that have been made based primarily on theoretical considerations.
3. To provide information on how users of elliptic curve public-key cryptosystems should select suitable key lengths for a desired level of security.
4. To determine whether there is any significant difference in the difficulty of the ECDLP for elliptic curves over F_p.
5. To encourage and stimulate research in computational and algorithmic number theory and, in particular, the study of the ECDLP.

2 The Elliptic Curve Discrete Logarithm Problem (ECDLP)

This section provides a brief overview of the state-of-the-art in algorithms known for solving the elliptic curve discrete logarithm problem.

2.1 The discrete logarithm problem

Roughly speaking, the *discrete logarithm problem* is the problem of "inverting" the process of exponentiation. The problem can be posed in a variety of algebraic settings. The most commonly studied versions of this problem are:

1. ***The discrete logarithm problem in a finite field (DLP):*** Given a finite field F_q and elements $g, h \in \mathsf{F}_q$. find an integer l such that $g^l = h$ in F_q, provided that such an integer exists.
2. ***The elliptic curve discrete logarithm problem (ECDLP):*** Given an elliptic curve E defined over a finite field F_q and two points $P, Q \in E(\mathsf{F}_q)$, find an integer l such that $lP = Q$ in E, provided that such an integer exists

On the surface, these two problems look quite different. In the first problem, "multiplicative" notation is used: g^l refers to the process of *multiplying g* by itself l times. In the second problem, "additive" notation is used: lP refers to the process of *adding P* to itself l times.

If one casts these notational differences aside, then the two problems are abstractly the same. What is intriguing about the two problems, however, is that the second appears to be much more difficult than the first. The fundamental reason for this is that the algebraic objects in the DLP (*finite fields*) are equipped with two basic operations: addition and multiplication of field elements. In contrast, the algebraic objects in the ECDLP (*elliptic curves over finite felds*) are equipped with only one basic operation: addition of elliptic curve points. The additional structure present in the DLP has led to the discovery of the *index-calculus methods,* which have a *subexponential* running time. Elliptic curves do not possess this additional structure, and for this reason no one has been able to apply the index-calculus methods to the ECDLP (except in very special and well-understood cases). This absence of subexponential-time algorithms for the ECDLP, together with efficient implementation of the elliptic curve arithmetic, is precisely the reason that ECC have proven so attractive for practical use.

2.2 Algorithms known for the ECDLP

This section briefly overviews the algorithms known for the ECDLP. All of these algorithms take *fully exponential time.*

The following notations are used:

1 q is the order of the underlying finite field.

2 F_q is the underlying finite field of order q.

3 E is an elliptic curve defined over F_q.

4 $E(\mathsf{F}_q)$ is the set of points on E both of whose coordinates are in F_q , together with the point at infinity.

5 P is a point in $E(\mathsf{F}_q)$.

6 n is the large prime order of the point P.

7 Q is another point in $E(\mathsf{F}_q)$.

The ECDLP is: Given q, E, P, n and Q, find an integer l, $0 \leq l \leq n - 1$, such that $lP = Q$, provided that such an integer exists.

For the remainder of the discussion, we shall only consider instances of the ECDLP for which the integer l exists.

1. ***Naive exhaustive searc*****h**.

In this method, one simply computes successive multiples of P: P, *2*P, *3*P, *4*P, ... until Q is obtained. This method can take up to n steps in the worst case.

2. ***Baby-step giant-step algorith*****m**.

This algorithm is a time-memory trade-off of the method of exhaustive search. It requires storage for about $\sqrt{n}$ points, and its running time is roughly $\sqrt{n}$ steps in the worst case.

3 Pollard's rho algorithm.

This algorithm, due to Pollard, is a randomized version of the baby-step giant-step algorithm. It has roughly the same expected running time ($\sqrt{\pi n}/2$ steps) as the baby-step giant-step algorithm, but is superior in that it requires a negligible amount of storage., and Wiener and Zuccherato [7] showed how Pollard's rho algorithm can be sped up by a factor of 2 . Thus the expected running time of Pollard's rho method with this speedup is $\sqrt{\pi n/2}$ steps.

4. ***Distributed version of Pollard's rho algorith*****m**.

Pollard's rho algorithm can be parallelized so that when the algorithm is run in parallel on m processors, the expected running time of the algorithm is roughly $\sqrt{\pi n/(2m)}$ steps. That is, using m processors results in an m-fold speed-up.This distributed version of Pollard's rho algorithm is the fastest general-purpose algorithm known for the ECDLP.

5. ***Pohlig-Hellman algorith*****m**.

This algorithm reduces the problem of recovering l to the problem of recovering l modulo each of the prime factors of n; the desired number l can then be recovered by using the Chinese Remainder Theorem.The implications of this algorithm are the following. To construct the most difficult instance of the ECDLP, one must select an ellipticcurve whose order is divisible by a large prime n. Preferably, this order should be a prime or almost a prime (i.e. a large prime n times a small integer h). The elliptic curves in the exercises and challenges posed here are all of this type.

6. ***Pollard's lambda method***.

This is another randomized algorithm due to Pollard. Like Pollard's rho method, the lambda method can also be parallelized with a linear speedup. The parallelized lambda method is slightly slower than the plelized rho method. The lambda method is, however, faster in situations when the logarithm being sought is known to lie in a subinterval $[0,b]$ of $[0,n-1]$ where $b < 0.39n$.

7. ***Multiple Algorithms***

If a single instance of the ECDLP (for a given elliptic curve E and a base point P) is solved using Pollard's rho method, then the work done in solving this instance can be used to speed up the solution of other instance of the ECDLP for the same curve E and base point P. More precisely, solving k instances of the ECDLP (for the same curve E and base point P) takes only k as much work as it does to solve one instance of the ECDLP. This analysis, however, does not take into account storage requirements. Concerns that successive logarithms become easier can be addressed by ensuring that the elliptic parameters are chosen so that the first instance is infeasible to solve.

9. ***Another special class of elliptic curves: anomalous curve*****s.**

An *anomalous elliptic curve* over F_q is an ellipic curve over F_q which has exactly q points. The attack does not extend to any other classes of elliptic curves. Consequently, by verifying that the number of points on an elliptic does not equal the number of elements in the underlying field, one can easily ensure that the Smart-Satoh-Araki attack does not apply to a particular curve.

3 The Challenge Explained

This section gives an overview of some of the mathematics that is relevant to this challenge. The format for the challenge parameters presented in Section 3.1.3 is also explained.

3.1 Elliptic curves over F_p - format and examples

3.1.1 The finite field F_p

Let p be a prime number. The finite field F_p is comprised of the set of integers $\{0, 1, 2, \ldots, p-1\}$ with the following arithmetic operations:

1 *Addition*: If $a, b \in F_p$, then $a + b = r$, where r is the remainder when $a + b$ is divided by p and $0 \le r \le p - 1$. This is known as addition modulo p.

2 *Multiplication:* If $a, b \in F_p$, then $a * b = s$, where s is the remainder when $a * b$ is divided by p and $0 \le s \le p - 1$. This is known as multiplication modulo p.

3 *Inversion:* If a is a non-zero element in F_p, the *inverse* of a modulo p, denoted a^{-1}, is the unique integer $c \in F_p$ for which $a * c = 1$.

Example (*The finite field* F_{23})

The elements of F_{23} are $\{0, 1, 2, \ldots, 22\}$. Examples of the arithmetic operations in F_{23} are:

1 $12 + 20 = 9$.

2 $8 * 9 = 3$.

3 $8^{-1} = 3$.

3.1.2 Elliptic curves over F_p

Let $p > 3$ be a prime number. Let $a, b \in F_p$ be such that $4a^3 + 27b^2 \neq 0$ in F_p. An *elliptic curve* $E(F_p)$ over F_p defined by the parameters a and b is the set of all solutions (x, y), $x, y \in F_p$, to the equation $y^2 = x^3 + ax + b$, together with an extra point O, the *point at infinity*.

The set of points $E(F_p)$ forms a group with the following addition rules:

1. $O + O = O$
2. $(x, y) + O = O + (x, y) = (x, y)$ for all $(x, y) \in E(F_p)$.
3. $(x, y) + (x, -y) = O$ for all $(x, y) \in E(F_p)$ (i.e., the negative of the point (x, y) is $-(x, y) = (x, -y)$).
4. (Rule for adding two distinct points that are not inverses of each other)

   Let $P = (x_1, y_1) \in E(F_p)$ and $Q = (x_2, y_2) \in E(F_p)$ be two points such that $x_1 \neq x_2$

   then $P + Q = (x_3, y_3)$,

   where $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = \lambda(x_1 - x_3) - y_1$

   and $\lambda = (y_2 - y_1) / (x_2 - x_1)$

5. (Rule for doubling a point)

   Let $P = (x_1, y_1) \in E(Fp)$ be a point with $y_1 \neq 0$. (If $y_1 = 0$ then $P = -P$, and so $2P = O$.)

then $2P=(x_3, y_3)$

where $x_{3=} \lfloor^2 - 2x_1$ and $y_3 = \lfloor(x_1 - x_3) - y_1$

and $\lfloor=(3x_1^2 + a)/2y_1$

Example *(An elliptic curve over* F_{23}*)*

$y^2 = x^3 + x + 1$ is an equation for an elliptic curve E over F_{23}. Here $a = 1$ and $b = 1$. The solutions over F_{23} to this equation are:

(0, 1) (0, 22) (1, 7) (1, 16) (3, 10) (3, 13) (4, 0) (5, 4) (5, 19) (6, 4) (6, 19)
(7, 11) (7, 12) (9, 7) (9, 16) (11, 3) (11, 20) (12, 4) (12, 19) (13, 7) (13, 16)
(17, 3) (17, 20) (18, 3) (18, 20) (19, 5) (19, 18)

$E(F_{23})$ has 28 points, including the point at infinity O. The following are examples of the addition law:

1 (3, 10) + (9, 7) = (17, 20).

2 2(3, 10) = (7, 12).

Example(ECDLP)

Consider the group $E_{23}(9, 17)$. This is the group defined by the equation $y^2 \bmod 23 = (x^3 + 9x + 17) \bmod 23$. What is the discrete logarithm k of Q = (4, 5) to the base P = (16.5)? The brute-force method is to compute multiples of P until Q is found.

Thus

P = (16, 5); 2P = (20, 20); 3P = (14, 14); 4P = (19, 20); 5P = (13, 10); 6P = (7, 3); 7P = (8, 7); 8P (12, 17); 9P = (4, 5).

Because 9P = (4, 5) = Q, the discrete logarithm Q = (4, 5) to the base P = (16, 5) is k = 9. In a real application, k would be so large as to make the brute-force approach infeasible.

3.1.3 Format for challenge parameters

This subsection describes the conventions used for representing the challenge parameters for elliptic curves over F_p.

Challenge parameters

p — the order of the finite field; p is a prime number.

seedE — the seed that was used to generate the parameters a and b (see Algorithm 1 in Section 3.1.4).

a, b — the field elements which define the elliptic curve $E: y^2 = x^3 + ax + b$.

seedP — the seed that was used to generate the point P (see Algorithm 3 in Section 3.1.4).

x_P, y_P — the x- and y-coordinates of the base point P.

n — the order of the point P; n is a prime number.

h — the co-factor h (the number of points in $E(\mathsf{F}_p)$ divided by n).

seedQ — the seed that was used to generate the point Q (see Algorithm 3 in Section 3.1.4).

x_Q, y_Q — the x- and y-coordinates of the public key point Q.

Data formats

Integers are represented in hexadecimal, the rightmost bit being the least significant bit. Example: The decimal integer 123456789 is represented in hexadecimal as 075BCD15.

Field elements (of F_p) are represented as hexadecimal integers.

Seeds for generating random elliptic curves and random elliptic curve points (see Section 3.2.4) are 160-bit strings and are represented in hexadecimal.

3.1.4 Random elliptic curves and points (the F_p case)

This subsection describes the method that is used for *verifiably* selecting elliptic curves and points at random. The defining parameters of the elliptic curve or point are defined to be outputs of the one-way hash function [6]

(as specified in [6]). The input seed to [6] then serves as proof (under the assumption that [6] cannot be inverted) that the elliptic curve or point were indeed generated at random.

The following notation is used: $t = \lceil \log_2 p \rceil$, $s = \lfloor (t - 1)/160 \rfloor$ and $h = t - 160 * s$.

Algorithm 1: Generating a random elliptic curve over F_p

Input: A field size p, where p is a prime.

Output: A 160-bit bit string seedE and field elements $a,b \in \mathsf{F}_p$ which define an elliptic curve E over F_p.

1. Choose an arbitrary bit string seedE of length 160 bits.
2. Compute H = [6](seedE), and let c_0 denote the bit string of length h bits obtained by taking the h rightmost bits of H.
3. Let W_0 denote the bit string of length h bits obtained by setting the leftmost bit of c_0 to 0.

   (This ensures that $r < p$.)
4. For i from 1 to s do:

   Compute W_i = [6]((seedE + i) mod 2^{160}).
5. Let W be the bit string obtained by the concatenation of W_0, W_1, …, W_s as follows: $W = W_0 \quad W_1 \quad \ldots \quad W_s$.
6. Let $w_1, w_2, \ldots, w_t$ be the bits of W from leftmost to rightmost. Let r be the integer

$$r = \sum_{i=1}^{i} w_i \, 2^{i-1}$$

7. Choose arbitrary integers $a, b \in \mathsf{F}_p$ such that $r * b^2 \equiv a^3 \pmod{p}$.

8. If $4a^3 + 27\,b^2 \equiv 0 \pmod{p}$ then go to step 1.

9. The elliptic curve chosen over F_p is $E : y^2 = x^3 + ax + b$.

10. Output(seedE, *a, b*).

Algorithm 2: Verifying that an elliptic curve was randomly generated

Input: A field size p (a prime), a bit string seedE of length 160 bits, and field elements $a, b \in \mathsf{F}_p$ which define an elliptic curve $E : y^2 = x^3 + ax + b$ over F_p.

Output: Acceptance or rejection that E was randomly generated using Algorithm 1.

1. Compute H = [6] (seedE), and let c_0 denote the bit string of length h bits obtained by taking the h rightmost bits of H.

2. Let W_0 denote the bit string of length h bits obtained by setting the leftmost bit of c_0 to 0.

3. For i from 1 to s do:

   Compute W_i = [6]((seedE + i) mod 2^{160}).

4. Let W' be the bit string obtained by the concatenation of $W_0, W_1, \ldots, W_s$ as follows: $W' = W_0 \quad W_1 \quad \ldots \quad W_s$.

5. Let $w_1, w_2, \ldots, w_t$ be the bits of W' from leftmost to rightmost. Let r' be the integer

$$r' = \sum_{i=1}^{i} w_i \, 2^{i-1}$$

5 If $r' * b^2 \equiv a^3 \pmod{p}$ then accept; otherwise reject.

Algorithm 3: Generating a random elliptic curve point

Input: Field elements $a, b \in \mathsf{F}_p$ which define an elliptic curve $E : y^2 = x^3 + ax + b$ over F_p. The order of $E(\mathsf{F}_p)$ is $n * h$, where n is a prime.

Output: A bit string seedP, a field element y_U, and a point $P \in E(\mathsf{F}_p)$ of order n.

1. Choose an arbitrary bit string seedP of length 160 bits.

2. Compute H = [6] (seedP), and let c_0 denote the bit string of length h bits obtained by taking the h rightmost bits of H.

3. Let x_0 denote the bit string of length h bits obtained by setting the leftmost bit of c_0 to 0.

4. For i from 1 to s do:

   Compute x_i = [6] ((seedE + i) mod 2^{160}).

5. Let x_U be the bit string obtained by the concatenation of $x_0, x_1, \ldots, x_s$ as follows: $x_u = x_0 \quad x_1 \quad \ldots \quad x_5$.

6. If the equation $y^2 = x^3{}_U + ax_U + b$ does not have a solution $y \in \mathsf{F}_p$, then go to step 1.

7. Select an arbitrary solution $y_U \in \mathsf{F}_p$ to the equation $y^2 = x^3{}_U + ax_U + b$

8. Let U be the point (x_U, y_U).

9. Compute $P = hU$.

10. If $P = \mathsf{O}$ then go to step 1.

11. Output(seedP, y_U, P).

Algorithm 4: Verifying that an elliptic curve point was randomly generated

Input: A field size p (a prime), field elements $a, b \in \mathsf{F}_p$ which define an elliptic curve $E : y^2 = x^3 + ax + b$ over F_p, a bit string seedP of length 160 bits, a field element $y_U \in \mathsf{F}_p$, and an elliptic curve point $P = (x_P, y_P)$. The order of $E(\mathsf{F}_p)$ is $n * h$, where n is a prime.

Output: Acceptance or rejection that P was randomly generated using Algorithm 3.

1. Compute $H = [6]$ (seedP), and let c_0 denote the bit string of length h bits obtained by taking the h rightmost bits of H.
2. Let x_0 denote the bit string of length h bits obtained by setting the leftmost bit of c_0 to 0.
3. For i from 1 to s do:
   Compute $x_i = [6]((\text{seedE} + i) \bmod 2^{160})$.
4. Let x_U be the bit string obtained by the concatenation of $x_0, x_1, \ldots, x_s$ as follows: $x_u = x_0 \quad x_1 \quad \ldots \quad x_5$
5. Let U be the point (x_U, y_U).
6. Verify that U satisfies the equation $y^2 = x^3 + ax + b$.
7. Compute $P' = hU$.
8. If $P \neq P'$ then reject.
9. Accept.

3.2 Further details about the challenge

This subsection presents some more information about the challenge. Each problem posed is to compute the private key given the elliptic curve parameters, the base point P of order n, and the public key point Q. The *private key* is the *unique* integer l, $0 \leq l \leq n - 1$, such that $Q = lP$. Each problem is therefore an instance of the elliptic curve discrete logarithm problem (ECDLP); see Section 2.

Here elliptic curves have been chosen randomly in a *verifiable* manner (see Sections 3.1.4) — anyone can verify that the elliptic curve parameters were indeed generated at random.

Another interesting feature of the challenge is that the points P and Q having order n were also chosen randomly in a *verifiable* manner. This means that each particular private key l is presently unknown even to the creators of the challenge!! However, any alleged solution l' that is found to a challenge can easily be verified by checking that $Q = l'P$.

The problems is divided into three sub-categories:

(i) Exercises

(ii) Level I Challenges , and

(iii) Level II Challenges.

These are distinguished by the size of the parameter *n,* the prime order of the base point *P*. As the size of *n* increases, the problem is expected to become harder. By a *k-bit challenge,* we shall mean a challenge whose parameter *n* has bit length *k*.

3.3 Time estimates for exercises and challenges

This subsection provides a *very rough* estimate for the time to solve a *k*-bit challenge with parameter *n.* These estimates are for software implementations; we do not assume that any special hardware for parallelized Pollard rho attacks is used.

Recall from Section 2.2 that the distributed version of Pollard's rho algorithm using *M* processors takes approximately $\sqrt{\pi n/2}/M$ steps. Here, each "step" is an elliptic curve addition or double together with some rho-method specific operations such as evaluations of hash functions and/or a membership test.

Thus, if a computer can perform *l* operations per second, then the number of computer days required before a discrete logarithm is found is expected to be roughly

$$(1 / (l * 60 * 60 * 24)) * (\sqrt{\pi n} / 2 M) = 10^{-5}(\sqrt{n} / l M)$$

for all other curves.

When estimating the challenge problems, we also take into account that the iterations scale quadratically on the number of machine words required by the field. We assume that we work with a 32-bit machine. Then, for example, a 109-bit field requires 4 machine words while a 89-bit field requires only 3 machine words. This means that each iteration in a 109-bit field should cost $(4/3)^2$ as much.

For a curve over a 89-bit *prime* field, we estimate that a Pentium 100 can perform about $l = 48000$ iterations per second. Here we assume that iterations are done on orbits of points rather that on points. This slightly increases the time needed for one iteration while it considerably reduces the expected number of iterations.

The 109-bit Level I challenges are feasible using a very large network of computers. The 131-bit Level I challenges are expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered.

The Level II challenges are infeasible given today's computer technology and knowledge. The elliptic curves for these challenges meet the stringent security requirements imposed by existing and forthcoming ANSI banking standards.

An implementation report of the Pollard rho algorithm for solving the ECDLP can be found in. An

implementation report of the solution of some of the exercises can be found in .

The estimates are included in the tables in section 5.

4 Exercise Lists and Challenge Lists

4.1 Elliptic curves over F_p

In the following tables, ECCp-*k* denotes that the exercise or challenge is over a field F_p (*p* rime), and that the parameter *n* has bit length *k*.

For a description of the format of the challenge parameters, see Section 3.1.3. For further details about the challenge, see Section 3.3. The time estimates for each exercise and challenge were derived as in Section 3.4.

Using these timings, it is expected that the 79-bit exercise could be solved in a matter of hours, the 89-bit exercise could be solved in a matter of days, and the 97-bit exercise in a matter of weeks using a network of 3000 computers.

The 109-bit Level I challenge is feasible using a very large network of computers. The 131-bit Level I challenge is expected to be infeasible against realistic software and hardware attacks, unless of course a new algorithm for the ECDLP is discovered.

The Level II challenges are infeasible given today's computer technology and knowledge. The elliptic curves for these challenges meet the stringent security requirements imposed by existing and forthcoming .

4.1.1 Exercises

Exercise	Field size(in bits)	Estimated number of machine days
ECCp-79	79	146
ECCp-89	89	4360
ECCp-97	97	71982

4.1.2 Level I challenges

Exercise	Field size(in bits)	Estimated number of machine days
ECCp-109	109	9.0×10^6
ECCp-131	131	2.3×10^{10}

4.1.3 Level II challenges

Exercise	Field size(in bits)	Estimated number of machine days
ECCp-163	163	2.3×10^{15}
ECCp-191	191	4.8×10^{19}
ECCp-239	239	1.4×10^{27}
ECCp-359	359	3.7×10^{45}

5 Status of the ECC Challenge

The challenge was released on November 6, 1997, at 1 p.m. The table below shows which problems have been solved so far. Here, the date given as end date indicates the day of submission of the solution to Certicom.

5.1 Elliptic curves over F_p

Exercises						
				Estimated number of		
Challenge	**Start Date**	**End Date**	**Number of Elliptic Curve Operations taken**	**Elliptic Curve Operations**	**Iterations Per second***	**Machine days***
ECCp-79	Nov. 6,1997	Dec.16, 1997	$1.4x10^{12}$	$6.1x10^{11}$	48000	146
ECCp-89	Nov. 6,1997	Jan. 12, 1998	$2.4x10^{13}$	$1.8x10^{13}$	48000	4360
ECCp-97	Nov. 6,1997	Mar.18, 1998	$2.0x10^{14}$	$3.0x10^{14}$	48000	71982

Level I Challenges						
				Estimated number of		
Challenge	**Start Date**	**End Date**	**Number of Elliptic Curve Operations taken**	**Elliptic Curve Operations**	**Iterations Per second***	**Machine days***
ECCp-109	Nov.6,1997			$2.1x10^{16}$	27000	$9.0x10^{6}$
ECCp-131	Nov.6,1997			$3.5x10^{19}$	17820	$2.3x10^{11}$

Level II Challenges						
				Estimated number of		
Challenge	**Start Date**	**End Date**	**Number of Elliptic Curve Operations taken**	**Elliptic Curve Operations**	**Iterations Per second***	**Machine days***
ECCp-163	Nov.6,1997			$2.4x10^{24}$	12000	$2.3x10^{15}$
ECCp-191	Nov.6,1997			$4.9x10^{28}$	12000	$4.8x10^{19}$
ECCp-239	Nov.6,1997			$8.2x10^{35}$	6750	$1.4x10^{27}$
ECCp-359	Nov.6,1997			$9.6x10^{53}$	3000	$3.7x10^{45}$

*Estimation based on a Pentium 100

5.2 Details of solved problems

5.2.1 ECCp-79

Solution: $\log_P Q$ = 1387 56822DD5 FB093766

Date: 6 Dec 1997

Who solved it: W. Baisley and R.J. Harley

Method: Pollard's rho method, parallelized

Brent-type cycle-finding algorithm:

About 1 400 000 000 000 iterations

52.2 ECCp-89

Solution: $\log_P Q$ = 0113C284 D9BD7B58 BCA30C67

Date: 12 Jan 1998

Who solved it: group of 57 people

Method: parallelized Pollard rho method:

24 249 418 904 337 iterations

36 345 distinguished points

5.2.3 ECCp-97

Solution: $\log_P Q$ = 01 6C86AA7C ACF69F1D D28B3E2F

Date: 18 Mar 1998

Who solved it: group of 588 people

Method: parallelized Pollard rho method

About 2.0 x 10^{14} iterations

186 364 distinguished points

6 Conclusion

ECDLP is the inverse operation in the Elliptic Curve Cryptosystem -- i.e we have to perform to get the plaintext back from the ciphertext, given only the public key. In section 3.2 if we use a large enough prime field, then the number of possible values for *l* becomes inconveniently large. So inconveniently large that it 's quite practical to create a sufficiently large prime field that searching through the possible values of *l* would take much more processor time. ECDLP is significantly more difficult than DLP. The general conclusion is that the ECDLP in fact requires fully exponential time to solve.

7 Further Works

In future there is vast scope of research regarding Elliptic Curve Cryptography. At first I have to understand a lot of things about the challenges and also about the solved challenges. After that I want to solve the challenges of the ECC using modern computer technology in small numbers of node present in a network. And also want to check the performance of the challenges.

8 References

Books

1. "Cryptography & Network Security Principals & Practices" by William Stalings
2. "Guide to ECC " by Darvel Hankerson.

Websites

1. www.certicom.com
2. www.cryptoman.com
3. www.IEEXplore.com

Research Paper

1 N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation,* volume 48, pages 203-209, 1987.

2 A. Menezes, *Elliptic Curve Pub1ic Key Cryptosystems,* Kluwer Academic Publishers,1993.

3 V. Miller, "Uses of elliptic curves in cryptography", *Advances in Cryptology - CRYPTO'85,*

4 A. Menezes, *Elliptic Curve Pub1ic Key Cryptosystems,* Kluwer Academic Publishers,1993.

5 Hankerson. Menezes. Vanstone. "Guide to Elliptic Curve Cryptography",Springer-Verlag 2004

6 FIPS 180-1, "Secure hash standard", Federal Information Processing Standards Publication 180-1, U.S. April 1995.

7 M. Wiener and R. Zuccherato, "Faster attacks on elliptic curve cryptosystems", *Selected Areas in Cryptograph*y,

BigInteger Fields , Methods and Constructor used in this program

java.math
Class BigInteger
java.lang.Object
  java.lang.Number
    java.math.BigInteger
All Implemented Interfaces:
Comparable, Serializable

---

public class BigInteger
extends Number
implements Comparable

//
Field Summary
//

static BigInteger ONE The BigInteger constant one.
static BigInteger ZERO The BigInteger constant zero.

Constructor Summary

BigInteger(String val)
Translates the decimal String representation of a BigInteger into a BigInteger.

//
Method Summary
//

BigInteger add(BigInteger val)
Returns a BigInteger whose value is (this + val).

int compareTo(BigInteger val)
Compares this BigInteger with the specified BigInteger.

BigInteger divide(BigInteger val)
Returns a BigInteger whose value is (this / val).

BigInteger mod(BigInteger m)
Returns a BigInteger whose value is (this mod m).

BigInteger modInverse(BigInteger m)
Returns a BigInteger whose value is (this-1 mod m).

BigInteger multiply(BigInteger val)
Returns a BigInteger whose value is (this * val).

BigInteger negate()
Returns a BigInteger whose value is (-this).

BigInteger pow(int exponent)
Returns a BigInteger whose value is (thisexponent).

static BigInteger probablePrime(int bitLength, Random rnd)
Returns a positive BigInteger that is probably prime, with the specified bitLength.

BigInteger remainder(BigInteger val)
Returns a BigInteger whose value is (this % val).

String toString()
Returns the decimal String representation of this BigInteger.

static BigInteger valueOf(long val)
Returns a BigInteger whose value is equal to that of the specified long.

//
Field Detail
//

ZERO
public static final BigInteger ZEROThe BigInteger constant zero.

ONE
public static final BigInteger ONEThe BigInteger constant one.

//
Method Detail
//

probablePrime

public static BigInteger probablePrime(int bitLength,Random rnd)Returns a positive BigInteger that is probably prime, with the specified bitLength. The probability that a BigInteger returned by this method is composite does not exceed 2-100.

Parameters:
bitLength - bitLength of the returned BigInteger.
rnd - source of random bits used to select candidates to be tested for primality.

Returns:
a BigInteger of bitLength bits that is probably prime
Throws:
ArithmeticException - bitLength < 2.
See Also:
bitLength()

---

valueOf
public static BigInteger valueOf(long val)Returns a BigInteger whose value is equal to that of the specified long. This "static factory method" is provided in preference to a (long) constructor because it allows for reuse of frequently used BigIntegers.

Parameters:
val - value of the BigInteger to return.
Returns:
a BigInteger with the specified value.

---

add
public BigInteger add(BigInteger val)Returns a BigInteger whose value is (this + val).

Parameters:
val - value to be added to this BigInteger.
Returns:
this + val

---

subtract
public BigInteger subtract(BigInteger val)Returns a BigInteger whose value is (this - val).

Parameters:
val - value to be subtracted from this BigInteger.
Returns:
this - val

---

multiply
public BigInteger multiply(BigInteger val)Returns a BigInteger whose value is (this * val).

Parameters:
val - value to be multiplied by this BigInteger.
Returns:
this * val

---

divide
public BigInteger divide(BigInteger val)Returns a BigInteger whose value is (this / val).

Parameters:
val - value by which this BigInteger is to be divided.
Returns:
this / val
Throws:
ArithmeticException - val==0

---

remainder
public BigInteger remainder(BigInteger val)Returns a BigInteger whose value is (this % val).

Parameters:
val - value by which this BigInteger is to be divided, and the remainder computed.
Returns:
this % val
Throws:
ArithmeticException - val==0

---

pow
public BigInteger pow(int exponent)Returns a BigInteger whose value is (thisexponent). Note that exponent is an integer rather than a BigInteger.

Parameters:
exponent - exponent to which this BigInteger is to be raised.
Returns:
thisexponent
Throws:
ArithmeticException - exponent is negative. (This would cause the operation to yield a non-integer value.)

---

negate
public BigInteger negate()Returns a BigInteger whose value is (-this).

Returns:
-this

---

mod
public BigInteger mod(BigInteger m)Returns a BigInteger whose value is (this mod m). This method differs from remainder in that it always returns a non-negative BigInteger.

Parameters:
m - the modulus.
Returns:
this mod m
Throws:
ArithmeticException - m <= 0

---

modInverse
public BigInteger modInverse(BigInteger m)Returns a BigInteger whose value is (this-1 mod m).

Parameters:
m - the modulus.
Returns:
this-1 mod m.
Throws:
ArithmeticException - m <= 0, or this BigInteger has no multiplicative inverse mod m (that is, this BigInteger is not relatively prime to m).

---

compareTo
public int compareTo(BigInteger val)Compares this BigInteger with the specified BigInteger. This method is provided in preference to individual methods for each of the six boolean comparison operators (<, ==, >, >=, !=, <=). The suggested idiom for performing these comparisons is: (x.compareTo(y) <op> 0), where <op> is one of the six comparison operators.

Parameters:
val - BigInteger to which this BigInteger is to be compared.
Returns:
-1, 0 or 1 as this BigInteger is numerically less than, equal to, or greater than val.

---

toString
public String toString()Returns the decimal String representation of this BigInteger. The digit-to-character mapping provided by Character.forDigit is used, and a minus sign is prepended if appropriate. (This representation is compatible with the (String) constructor, and allows for String concatenation with Java's + operator.)

Overrides:
toString in class Object

Returns:
decimal String representation of this BigInteger.

Classes and methods used in this project

keyGen :

Takes input a , b and prime field (m) .

Produces max order, public key, private key, G and total keyGen time . Encapsulates these information in data object and writes the data object in ‘data.doc’ . It takes help of ‘objectInputStream’ and ‘objectOutputStream’ .

ePoint :

Produces elliptic curve point with x-coordinate and y-coordinate .

data :

Encapsulates the data produced by keyGen by using ‘setData’ method .

Prototype of setdata
public void setData(BigInteger a,BigInteger b,BigInteger m,BigInteger n,ePoint G,ePoint[] take) .

encrypt :

Encrypts the plain text and produces the cipher text (using key stored in the ‘data.doc’ and cphrData object) . It takes help of ‘objectInputStream’ and ‘objectOutputStream’ .

cphrData :

It has a method setData . Prototype of this method shown below
public void setData(ePoint t[]) .

decrypt :

It decrypts the cipher text and produces the plain text in ‘recoveredText.txt’ .

keyGen.java

```
import java.util.Random;
import java.io.*;
import java.util.Calendar;
import java.math.BigInteger;


public class keyGen {

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

      BigInteger l=null,yy,xx,modx,mody;
```

```
        BigInteger m,a,b,maxorder,xp,xq,yp,yq,xr,yr,x,y,n,p;

        int odpos,i,j;

        ePoint point[]=new ePoint[257] ,take[]=new ePoint[256];
        for(i=0;i<257;i++) point[i]=new ePoint();
        for(i=0;i<256;i++) take[i]=new ePoint();

        long current_time, end_time;

        Random randObj=new Random();

        int range,rand=(int)(randObj.nextDouble()*(double)100);

        BufferedReader stdin=new BufferedReader(new
InputStreamReader(System.in));

        System.out.print("Enter the value of a in elliptic curve : ");
        a=new BigInteger(stdin.readLine());
        System.out.print("\nEnter the value of b in elliptic curve : ");
        b=new BigInteger(stdin.readLine());
        System.out.println("\nThe field range:
"+(m=BigInteger.probablePrime(16,randObj)).toString());

        // Getting current Time
      current_time=Calendar.getInstance().getTimeInMillis();
        range=2+rand%25;
        i=0;
        String r=String.valueOf(range);
        for(x=new BigInteger(r);x.compareTo(m)<=0;x=x.add(BigInteger.ONE))
        {
                        xx=x.pow(3).add(a.multiply(x)).add(b);
                        modx=xx.mod(m);

                for(y=new
BigInteger(r);y.compareTo(m)<=0;y=y.add(BigInteger.ONE))
                {
                        yy=y.multiply(y);
                        mody=yy.mod(m);

                        if(modx.equals(mody))
                        {
                                System.out.println("("+x+","+y+")");
                                point[i].setX(x);
                        point[i].setY(y);
                        i++;
                                break;
```

```
                        }

                }
                if(i>256)
                        break;
        }

        for(i=0;i<257;i++)
        {
                point[i].setOrder(BigInteger.ZERO);
        }
//for(i=0;i<257;i++)System.out.println("Points
"+point[i].getX()+","+point[i].getY()+","+point[i].getOrder());
        for(i=0;i<15;i++)
        {
                xp=point[i].getX();
                yp=point[i].getY();
                xq=xp;
                yq=yp;
                p=new BigInteger("1");
                while(true)
                {
                        BigInteger l1,l2;
                        p=p.add(BigInteger.ONE);
                        if(xp.equals(xq))
                        {
                                if(yp.equals(yq))
                                {

l1=BigInteger.valueOf(3).multiply(xp.pow(2)).add(a).mod(m);

l2=BigInteger.valueOf(2).multiply(yp).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
                                else
                                {
                                        point[i].setOrder(p);
                                        break;
                                }
                        }
                        else
                        {
                                if(xq.subtract(xp).compareTo(BigInteger.ZERO)>=0)
                                {
                                        l1=yq.subtract(yp).mod(m);
                                        l2=xq.subtract(xp).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
```

```
                                else
                                {
                                        l1=yp.subtract(yq).mod(m);
                                        l2=xp.subtract(xq).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
                        }
                        xr=l.pow(2).subtract(xp).subtract(xq).mod(m);
                        yr=l.multiply(xp.subtract(xr)).subtract(yp).mod(m);
                        xp=xr;
                        yp=yr;
                }
        }
//for(i=0;i<257;i++)System.out.println(point[i].getX()+","+point[i].getY()+","+point[i
].getOrder());
        maxorder=point[0].getOrder();
        odpos=0;
        for(i=1;i<15;i++)
        {
        if(point[i].getOrder().compareTo(maxorder)>0)
        {
                maxorder=point[i].getOrder();
                        odpos=i;
        }
        }
        System.out.println("\nMax Order : "+maxorder);
        n=maxorder.divide(new BigInteger("2")).subtract(new
BigInteger(String.valueOf(rand)).remainder(new BigInteger("25")));
System.out.println("\nn : "+n);
        xp=point[odpos].getX();
        yp=point[odpos].getY();
        xq=point[odpos].getX();
        yq=point[odpos].getY();
        p=new BigInteger("1");
        while(!p.equals(n))
        {
                BigInteger l1,l2;
                p=p.add(BigInteger.ONE);
                if(xp.equals(xq))
                {
                        if(yp.equals(yq))
                        {
                                l1=new
BigInteger("3").multiply(x.pow(2)).add(a).mod(m);
                                l2=new BigInteger("2").multiply(yp).modInverse(m);
                                l=l1.multiply(l2).mod(m);
                        }
                        else continue;
```

```
                }
                else
                {
                        if(xq.subtract(xp).compareTo(BigInteger.ZERO)>=0)
                        {
                                l1=yq.subtract(yp).mod(m);
                                l2=xq.subtract(xp).modInverse(m);
                                l=l1.multiply(l2).mod(m);
                        }
                        else
                        {
                                l1=yp.subtract(yq).mod(m);
                                l2=xp.subtract(xq).modInverse(m);
                                l=l1.multiply(l2).mod(m);
                        }
                }
                xr=l.pow(2).subtract(xp).subtract(xq).mod(m);
                yr=l.multiply(xp.subtract(xr)).subtract(yp).mod(m);
                xp=xr;
                yp=yr;
        }

        if(p.equals(n))
                System.out.println("\nYour PUBLIC KEY is=("+xp+","+yp+")");


        j=0;
        for(i=0;i<257;i++)
        {
                if(i==odpos)
                        continue;
                take[j].setX(point[i].getX());
                take[j].setY(point[i].getY());
                take[j].setOrder(point[i].getOrder());
                j++;
        }

        //file code 2 b written
        System.out.println("\nYour PRIVATE KEY is : "+n);

// Getting end Time
        end_time=Calendar.getInstance().getTimeInMillis();

        ePoint G=new ePoint();
        G.setX(point[odpos].getX());
        G.setY(point[odpos].getY());
        G.setOrder(point[odpos].getOrder());
```

```
        System.out.println("Your 'G' is ("+G.getX()+","+G.getY()+")");

        data D=new data();
        D.setData(a,b,m,n,G,take);

        try{
        ObjectOutputStream o=new ObjectOutputStream(new
FileOutputStream("data.doc"));
        o.writeObject(D);
        o.flush();
        o.close();
        }
        catch(Exception e){
                System.out.println("Error while writting data object !");
        }

        System.out.println("\nStart "+current_time+" MSec ");
      System.out.println("\nEnd "+end_time+" MSec  ");

      System.out.println("\nTotal keyGen Time = "+(end_time-current_time)+" MSec
\n");

} // End of main



} //End of class keyGen

                                        class ePoint implements Serializable
                                {
                                        public ePoint()
                                        {

                                        }
                                        private BigInteger xcoord;
                                        private BigInteger ycoord;
                                                        private BigInteger order;

                                        public void setX(BigInteger x)
                                        {
                                                xcoord=x;
                                        }

                                                 public void setY(BigInteger y)
                                        {
                                                ycoord=y;
                                        }
```

```
                                        public void setOrder(BigInteger o)
                                {
                                        order=o;
                                }

                                public BigInteger getX()
                                {
                                        return xcoord;
                                }

                                public BigInteger getY()
                                {
                                        return ycoord;
                                }

                                public BigInteger getOrder()
                                {
                                        return order;
                                }

                        }


        class data implements Serializable
        {
        public BigInteger a,b,m,n;
        public ePoint G,take[];

        public void setData(BigInteger a,BigInteger b,BigInteger m,BigInteger
n,ePoint G,ePoint[] take)
                {
                this.a=a;
                this.b=b;
                this.m=m;
                this.n=n;
                this.G=G;
                this.take=take;
                }
        }
```

encrypt.java

```
import java.io.*;
import java.util.Calendar;
import java.math.BigInteger;

public class encrypt{

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

 BigInteger x1,y1,xp,yp,xq,yq,i,p,q,s1,s2,xf,yf,xpb,ypb,xq1,yq1,xq2,yq2,xr,yr,k,l;
 ePoint take[]=new ePoint[256],G;
 data D=null;
 BufferedReader stdin=new BufferedReader(new InputStreamReader(System.in));
 long current_time, end_time;

        byte z;

        ePoint pt=new ePoint();

        try{
        ObjectInputStream o=new ObjectInputStream(new
FileInputStream("data.doc"));
        D=(data)o.readObject();
        o.close();
```

```
        }
        catch(Exception e){
                        System.out.println("Error while reading data object !");
        }

BigInteger      a=D.a,
                b=D.b,
                m=D.m,
                n=D.n;
        G=D.G;
        take=D.take;

/* Public key Entry */
      System.out.print("\n Enter x coordinate of Public key (receiver) : ");
        xpb=new BigInteger(stdin.readLine());
        System.out.print("\n Enter y coordinate of Public key (receiver) : ");
        ypb=new BigInteger(stdin.readLine());


/* checking whether the public key is on the curve or not */

        /*p=xpb.pow(3).add(a.multiply(xpb)).add(b);
                q= ypb.multiply(ypb);

                s1=p.mod(m);
                s2=q.mod(m);

                if(!s1.equals(s2))
                {
                System.out.println("\n\n Wrong entry!\nThe public is not on the curve
");
                System.exit(0);
                }*/


        /* random number selection */

        System.out.print("\n Enter the random number( less than "+n+" ) selected by
sender: ");
        k=new BigInteger(stdin.readLine());

        /* plaintext entry */

        InputStream in=new FileInputStream("PlainText.txt");

        //System.out.println(in.available());

        /* Encryption */
```

```
// Getting current Time

current_time=Calendar.getInstance().getTimeInMillis();


/* Making first point of the Ciphertext */

xp=G.getX();
yp=G.getY();
xq=xp;
yq=yp;
l=null;
p=new BigInteger("1");
while(!p.equals(k))
{

        BigInteger l1,l2;
        p=p.add(BigInteger.ONE);
        if(xp.equals(xq))
        {
                if(yp.equals(yq))
                {

l1=BigInteger.valueOf(3).multiply(xp.pow(2)).add(a).mod(m);
                        l2=BigInteger.valueOf(2).multiply(yp).modInverse(m);
                        l=l1.multiply(l2).mod(m);
                }
                else
                {
                        System.out.println("\n\n Can not be determinable ");
                        System.exit(0);
                }
        }
        else
        {
                if(xq.subtract(xp).compareTo(BigInteger.ZERO)>=0)
                        {
                                l1=yq.subtract(yp).mod(m);
                                l2=xq.subtract(xp).modInverse(m);
                                l=l1.multiply(l2).mod(m);
                        }
                        else
                        {
                                l1=yp.subtract(yq).mod(m);
                                l2=xp.subtract(xq).modInverse(m);
                                l=l1.multiply(l2).mod(m);
                        }
```

```
                }
                        xr=l.pow(2).subtract(xp).subtract(xq).mod(m);
                        yr=l.multiply(xp.subtract(xr)).subtract(yp).mod(m);
                        xp=xr;
                        yp=yr;
        }
        x1=xp;
        y1=yp;

        ObjectOutputStream out=new ObjectOutputStream(new
FileOutputStream("CipherText.txt"));
        pt.setX(x1);
        pt.setY(y1);
        ePoint t[]=new ePoint[in.available()+1];
        int s=0;
        t[s]=pt;

      /* Making second point of the Cipher Text */

        xp=xpb;
        yp=ypb;
        xq=x1;
        yq=y1;
        BigInteger l1,l2;
        p=new BigInteger("1");
        if(xp.equals(xq))
        {
                if(yp.equals(yq))
                {
                        l1=new BigInteger("3").multiply(xp.pow(2)).add(a).mod(m);
                        l2=new BigInteger("2").multiply(yp).modInverse(m);
                        l=l1.multiply(l2).mod(m);
                }
                else
                {

                                System.out.println("\n\n Can not be determinable ");
                                System.exit(0);
                }
        }
        else
        {
                        if(xq.subtract(xp).compareTo(BigInteger.ZERO)>=0)
                                {
                                        l1=yq.subtract(yp).mod(m);
                                        l2=xq.subtract(xp).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
```

```
                        else
                        {
                                l1=yp.subtract(yq).mod(m);
                                l2=xp.subtract(xq).modInverse(m);
                                l=l1.multiply(l2).mod(m);
                        }
        }
        xr=l.pow(2).subtract(xp).subtract(xq).mod(m);
        yr=l.multiply(xp.subtract(xr)).subtract(yp).mod(m);
        xp=xr;
        yp=yr;
        xq=xp;
        yq=yp;

        while((z=(byte)in.read())!=-1)
        {
                xp=take[z].getX();
                yp=take[z].getY();
      /* Checking whether the plaintext is on the curve or not */
                p=xp.pow(3).add(a.multiply(xp)).add(b);;
                q= yp.multiply(yp);

                s1=p.mod(m);
                s2=q.mod(m);

                if(!s1.equals(s2))
                {
                System.out.println("\n\n Wrong entry!\nThe point is not on the curve
");
                System.exit(0);
                }

                if(xp.equals(xq))
                {
                        if(yp.equals(yq))
                        {
                         l1=new BigInteger("3").multiply(xp.pow(2)).add(a).mod(m);
                         l2=new BigInteger("2").multiply(yp).modInverse(m);
                         l=l1.multiply(l2).mod(m);
                        }
                        else
                        {

                                System.out.println("\n\n Can not be determinable ");
                                System.exit(0);
                        }
                }
                else
```

```
                {
                        if(xq.subtract(xp).compareTo(BigInteger.ZERO)>=0)
                                {
                                        l1=yq.subtract(yp).mod(m);
                                        l2=xq.subtract(xp).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
                                else
                                {
                                        l1=yp.subtract(yq).mod(m);
                                        l2=xp.subtract(xq).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
                }
                pt.setX(l.pow(2).subtract(xp).subtract(xq).mod(m));
                pt.setY(l.multiply(xp.subtract(pt.getX())).subtract(yp).mod(m));
System.out.println(t[s].getX().toString()+","+t[s].getY().toString());
                t[++s]=pt;
        }
cphrData cd=new cphrData();
cd.setData(t);
        out.writeObject((cphrData)cd);
        end_time=Calendar.getInstance().getTimeInMillis();

        in.close();
        out.close();
// Getting end Time
      System.out.println("\nStart "+current_time+" MSec ");
      System.out.println("\nEnd "+end_time+" MSec  ");

      System.out.println("\nTotal encryption Time = "+(end_time-current_time)+"
MSec \n");

} // End of main
} //End of encrypt
class cphrData implements Serializable{
public ePoint []t;
public void setData(ePoint t[]){
this.t=t;
}
}
```

decrypt.java

```
import java.io.*;
import java.util.Calendar;
import java.math.BigInteger;

public class decrypt{

public static void main(String[] args)throws IOException{
        BigInteger p,xq,yq,xr,yr,xp,yp,l,x1,y1,pk,xt,yt;
        ePoint pt[],take[]=new ePoint[256];
        BufferedReader stdin=new BufferedReader(new
InputStreamReader(System.in));
        long current_time, end_time;
        data D=null;

        try{
        ObjectInputStream o1=new ObjectInputStream(new
FileInputStream("data.doc"));
        D=(data)o1.readObject();
        o1.close();
        }
        catch(Exception e)
        {
        System.out.println("Error on opening data document !");
        }


 BigInteger     a=D.a,
                b=D.b,
                m=D.m,
                n=D.n;
        take=D.take;
```

```
        System.out.print("\n Enter your private key: ");
        pk=new BigInteger(stdin.readLine());

    // Getting current Time

        current_time=Calendar.getInstance().getTimeInMillis();

        xp=D.G.getX();
        yp=D.G.getY();
        xq=D.G.getX();
        yq=D.G.getY();
        p=new BigInteger("1");
        BigInteger l1,l2;
        l=null;
        while(!p.equals(pk))
        {
                p=p.add(BigInteger.ONE);
                if(xp.equals(xq))
                {
                        if(yp.equals(yq))
                        {


l1=BigInteger.valueOf(3).multiply(xp.pow(2)).add(a).mod(m);
                                l2=BigInteger.valueOf(2).multiply(yp).modInverse(m);
                                l=l1.multiply(l2).mod(m);
                        }
                        else
                        {
                                System.out.println("\n\n Can not be determinable ");
                                System.exit(0);
                        }
                }
                else
                {               if(xq.subtract(xp).compareTo(BigInteger.ZERO)>=0)
                                {
                                        l1=yq.subtract(yp).mod(m);
                                        l2=xq.subtract(xp).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
                                else
                                {
                                        l1=yp.subtract(yq).mod(m);
                                        l2=xp.subtract(xq).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
                }
```

```
                    xr=l.pow(2).subtract(xp).subtract(xq).mod(m);
                    yr=l.multiply(xp.subtract(xr)).subtract(yp).mod(m);
                    xp=xr;
                    yp=yr;
        }

        ObjectInputStream in=new ObjectInputStream(new
FileInputStream("CipherText.txt"));
        InputStream is=new FileInputStream("PlainText.txt");
        cphrData cd=new cphrData();

        pt=new ePoint[is.available()+1];
        int s=0;
        try{
        cd=(cphrData)in.readObject();
        }
        catch(Exception e){
        System.out.println("Class not found-1 !");
        }
        pt=cd.t;
System.out.println(pt[s].getX().toString()+"HI");
        xq=pt[s].getX();
        yq=pt[s++].getY();

        if(xp.equals(xq))
        {
        if(yp.equals(yq))
           {



l1=BigInteger.valueOf(3).multiply(xp.pow(2)).add(a).mod(m);
                        l2=BigInteger.valueOf(2).multiply(yp).modInverse(m);
                        l=l1.multiply(l2).mod(m);
           }
           else
           {
                System.out.println("\n\n Can not be determinable ");
                    System.exit(0);
           }
        }
     else
     {
                        if(xq.subtract(xp).compareTo(BigInteger.ZERO)>=0)
                        {
                            l1=yq.subtract(yp).mod(m);
                            l2=xq.subtract(xp).modInverse(m);
                            l=l1.multiply(l2).mod(m);
```

```
                                }
                                else
                                {
                                        l1=yp.subtract(yq).mod(m);
                                        l2=xp.subtract(xq).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
        }

                        xr=l.pow(2).subtract(xp).subtract(xq).mod(m);
                        yr=l.multiply(xp.subtract(xr)).subtract(yp).mod(m);
                        xp=xr;
                        yp=yr;
        xq=xp;
        yq=yr.negate();
        OutputStream out=new FileOutputStream("recoveredText.txt");
        try{
        while(s<is.available()+1)
        {
                xp=pt[s].getX();
                yp=pt[s].getY();
System.out.println(xp.toString()+","+yp.toString());

                if(xp.equals(xq))
                {
                        if(yp.equals(yq))
                        {


l1=BigInteger.valueOf(3).multiply(xp.pow(2)).add(a).mod(m);
                                l2=BigInteger.valueOf(2).multiply(yp).modInverse(m);
                                l=l1.multiply(l2).mod(m);
                        }
                        else
                        {
                                System.out.println("\n\n Can not be determinable ");
                                System.exit(0);
                        }
                }
                else
                {

                                if(xq.subtract(xp).compareTo(BigInteger.ZERO)>=0)
                                {
                                        l1=yq.subtract(yp).mod(m);
                                        l2=xq.subtract(xp).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
```

```
                                else
                                {
                                        l1=yp.subtract(yq).mod(m);
                                        l2=xp.subtract(xq).modInverse(m);
                                        l=l1.multiply(l2).mod(m);
                                }
                        }

                        xt=l.pow(2).subtract(xp).subtract(xq).mod(m);
                        yt=l.multiply(xp.subtract(xt)).subtract(yp).mod(m);
                        for(int i=0;i<256;i++)
                        {
                                if(xt.equals(take[i].getX()) && yt.equals(take[i].getY()))
                                {
                                        out.write((char)i);
                                }
                        }
                s++;

                }
}
catch(Exception e){
System.out.println("Class not found-2 !");
}

                in.close();
                out.close();
                is.close();

// Getting end Time
                end_time=Calendar.getInstance().getTimeInMillis();

                System.out.println("\nStart "+current_time+" MSec ");
              System.out.println("\nEnd "+end_time+" MSec  ");

              System.out.println("\nTotal keyGen Time = "+(end_time-current_time)+" MSec
\n");

} // End of main

} //End of decrypt
```

RESULT

KEYGEN RESULT 1

Prime Field (m)=59887
Max Order : 59888

n : 29936

Your PUBLIC KEY is=(27,39409)

Your PRIVATE KEY is : 29936
Your 'G' is (27,20478)

Start 1211958010203 MSec

End 1211958026015 MSec

Total keyGen Time = 15812 MSec

KEYGEN RESULT 2

Prime Field (m)=45691
Max Order : 13480

n : 6723

Your PUBLIC KEY is=(45398,40292)

Your PRIVATE KEY is : 6723
Your 'G' is (19,22950)

Start 1211958824031 MSec

End 1211958852078 MSec

Total keyGen Time = 28047 MSec

KEYGEN RESULT 3

Prime Field (m)=44179
Max Order : 44180

n : 22079

Your PUBLIC KEY is=(27612,21310)

Your PRIVATE KEY is : 22079
Your 'G' is (13,21629)

Start 1211958984156 MSec

End 1211959010375 MSec

Total keyGen Time = 26219 MSec

KEYGEN RESULT 4

Prime Field (m)=52769
Max Order : 6568
n : 3278
Your PUBLIC KEY is=(52744,30869)
Your PRIVATE KEY is : 3278
Your 'G' is (9,326)
Start 1211959234953 MSec
End 1211959258984 MSec
Total keyGen Time = 24031 MSec

KEYGEN RESULT 5

Prime Field (m)=4005091
Max Order : 4005092
n : 2002545
Your PUBLIC KEY is=(3693595,3244191)
Your PRIVATE KEY is : 2002545
Your 'G' is (3,398600)
Start 1211958239468 MSec

End 1211959882437 MSec
Total keyGen Time = 1642969 MSec

//

ENCRYPTION RESULT 1

_____

File size 1kb

Start 1211959588265 MSec

End 1211959588453 MSec

Total encryption Time = 188 MSec

ENCRYPTION RESULT 2

_____

File size 9kb

Start 1211959778937 MSec

End 1211959779421 MSec

Total encryption Time = 484 MSec

ENCRYPTION RESULT 3

_____

File size 24kb
Start 1211959779426 MSec

End 1211959780710 MSec

Total encryption Time = 1284 MSec

//

DECRYPTION RESULT 1

_____

File size 1kb

Start 1211959588265 MSec

End 1211959588453 MSec

Total decryption Time = 188 MSec

DECRYPTION RESULT 2

File size 9kb

Start 1211959778940 MSec

End 1211959779421 MSec
Total decryption Time = 480 MSec

DECRYPTION RESULT 3

File size 24kb
Start 1211959779426 MSec
End 1211959780707 MSec
Total decryption Time = 1287 MSec