Cryptography and Elliptic Curves

Edmond Mbadu

Chestnut Hill College

Philadelphia, PA

A thesis submitted in fulfillment of the requirements for graduating in Mathematics and Computer Science at Chestnut Hill College

Director ____

Reader ____

Acknowledgements

To Professor Ryan Merilyn, thank you for supervising my senior thesis. Your articulate feedback was deeply appreciated.

To Professor Tammaro Elliot, I am grateful for your suggestions on my thesis.

To Professor Sullivan, thank you for providing me with such a wealth of resources that helped me start promptly.

To Professor Rody, thank you for giving me one of the best books on cryptography.

To Mr. Epp, I am grateful for your feedback and for your help on formatting my thesis.

To my classmates Ramon and Anya , thank you for your feedback.

To my family, thank you for supporting me in everything I do.

Table of Contents

1

Introduction

Cryptography is the art of hiding information. It consists of transforming messages into incomprehensible symbols called ciphertexts so that the unwanted eyes cannot comprehend them. The transformation of the message must be done in a way that is reversible for the intended parties to be able to communicate. Today, encryption algorithms, the methods used to transform messages, are almost always public. This is because publishing the encryption algorithm exposes it to an army of brilliant people ready to experiment on the algorithm and expose all the weaknesses in it. On top of that, hiding the encryption algorithm, as intuitive as it may seem, would not work most of the time. There may be a weakness in the structure of the algorithm that the creator of the encryption algorithm did not see which may be exploited by adversaries without the creator of the encryption algorithm knowing.

Much of modern cryptography uses mathematical theories. Also, given the computational complexity of encryption and decryption algorithms of the current age which mirror the present state of technology and science in general, computers are a must in the application of cryptography. Cryptography is then a hybrid; a field which combines both high level mathematics and computer science to achieve its goal. The history of cryptography is important in order to understand where cryptography is currently, how it has changed, and which events have accelerated its progress. [3, pp.3]

2

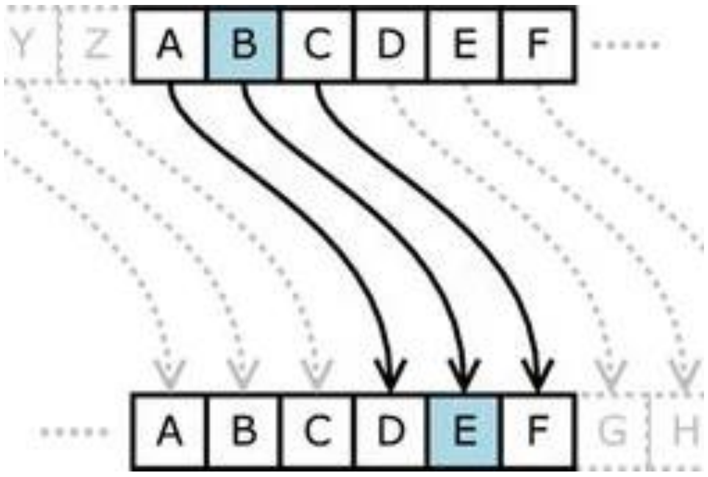
A Brief History of Cryptography

"History never really says goodbye. History says, See you later".

-Eduardo Galeano

One of the earliest and most famous uses of Cryptography can be traced back to Julius Caesar, emperor of Rome, who used it to communicate critical political messages. The encryption algorithm used by Caesar is today referred to as a Caesar cypher and belongs to a class of encryption algorithms called substitution ciphers. The enciphering method consists of mapping an individual letter to another letter in the same alphabet. For example, a plaintext letter is shifted down 3 letters, with a letter near the end of the alphabet wrapping around as shown in figure 1.1.

Figure 1.1



Hence, using this cipher, FIRE MISSILE would be enciphered as ILUH PLVVLOH.

Every Caesar cipher involves a shift and has a key. In this case, the key is k=3, also called the shift value. For an English based cipher, one may choose 0-25 as the shift value since there are only 26 letters in the English alphabet.

Of course, a Caesar cipher is easily breakable. To break the Caesar cipher, a method called "frequency analysis" is used. Frequency analysis exploits the fact that languages have letters that appear more often than others. In the English languages for instance, the letter ***e*** appears the most. Specifically, the letter ***e*** has a frequency of approximately12.7 percent, higher than any other letter in English. It is followed by ***t*** at 9 percent, and then **a** at 8 percent and so on. This fact can be used to find the shift value and break the code easily. Also, with current computational power, a brute force method, which consists of trying all the 26 keys, can be easily processed to break the code. [1]

The shift cipher may be generalized and slightly strengthened as follows. Choose a and b, with ***a and 26 being relatively prime***[a], and consider the function (called an *affine function*)

$$x \mapsto ax+b \pmod{26}$$

For instance, let $a = 7$ and $b = 8$, so we have $7x+8$. Consider plaintext letter such as c (=2). It is encrypted to $7\times 2 + 8 \equiv 22 \pmod{26}$, which is the letter w. Using the same function, the plaintext *cleopatra* encrypts to ***whkcjilxi***.[1]

[a] The reason *a* and 26 are required to be relatively prime is because if that is not the case, the affine function will not be one to one. In other words, two or more plaintext letters will be encrypted to the same ciphertext letter making it impossible to decrypt. In fact, there is a theorem that states that an affine function f(x)= ax+b (mod n) is one to one if and only if a and n are relatively prime.

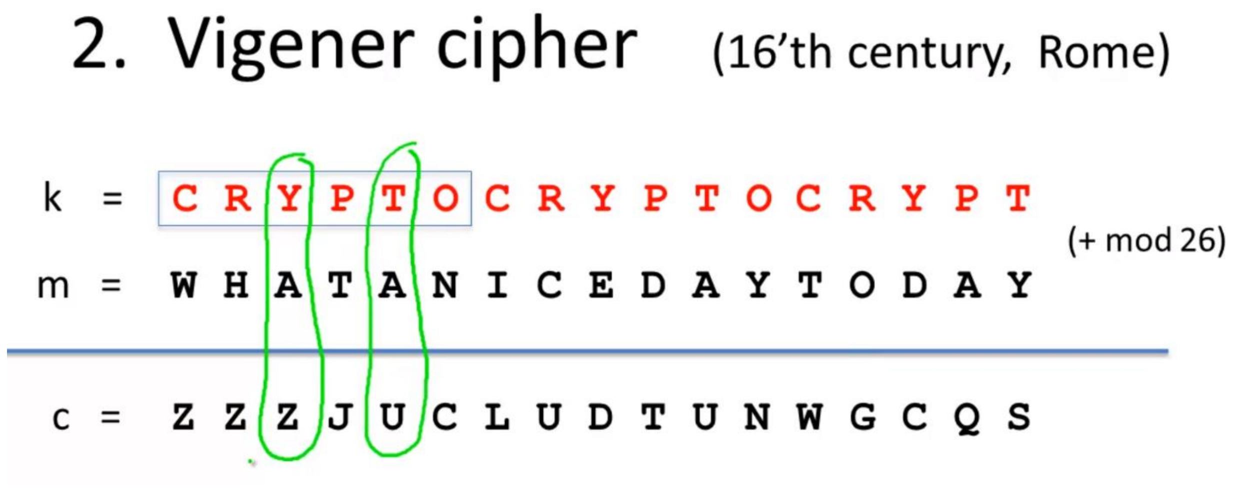
How do we decrypt the cipher? Decryption consists of solving the equation for *x*. Considering the function $y = 7x+8$, we obtain $x = \frac{1}{7}(y-8)$. However, $\frac{1}{7}$ needs to be reinterpreted when we work mod 26. In most cases finding the multiplicative inverse is easy. In this case however, we are looking for the multiplicative inverse mod n (26 for our case). Since 7 and 26 are coprime, there is a multiplicative inverse for 7 (mod 26)[b]. In fact, if the gcd(a, n)=1 and $ab \equiv ac$ (mod n), then $b \equiv c$ (mod n)[3, Chapter 3]. From that we can easily deduce that the multiplicative inverse of 7 is 15 mod 26. The decryption equation is then $x \equiv 15(y-8) \equiv 15y-120 \equiv 15y+10$ (mod 26). Unfortunately, given the current computational power, an exhaustive search through all the possible 312[c] keys can be used to break the affine cipher with relative ease.

Fast forward to the renaissance where a variation of the substitution cipher called *The Vigenère Cipher* was invented by Blaise de Vigenère, a French a cryptographer of the sixteenth century. The key for the encryption of a Vigenère cipher is a vector k of length L. For instance, let the vector k= (24,4,2,3,7,9,3) of length 6 be our key (figure 1.2). To encrypt a message using the given k, we take the first letter of the plaintext and shift it by 24. Then shift the second letter by 4, the third by 2, the fourth by 3, and so on. When the end of the key is reached, repeat the same operation until the entire message is covered (figure 1.2) The Vigenère Cipher was an achievement for its time. However, by the ninetieth century, Babbage and Kasiski, two prominent cryptographers of their time, had shown how to decrypt it. [2]

[b] To decide whether an equation of the form $a \cdot_n x = b$ has a unique solution in Z_n, it helps know whether a has a multiplicative inverse in Z_n, that is, whether there is another number a' such that $a' \cdot_n a = 1$. For example, in Z_9, the inverse of 2 is 5 because $2 \cdot_9 5 = 1$. On the other hand, 3 does not have an inverse in Z_9, because the equation $3 \cdot_9 x = 1$ does not have a solution. (This can be verified by checking the 9 possible values for x. It can be proven that the equation $a \cdot_n x = b$ has a solution if and only if the two values a and b are coprime).

[c] 312 possible keys since there are only 12 possible choices for *a* given that gcd(*a*, 26) =1, and 26 choices for *b* since we are only working mod 26

Fig 1.2


2. Vigener cipher (16’th century, Rome)
k = C R Y P T O C R Y P T O C R Y P T
m = W H A T A N I C E D A Y T O D A Y
c = Z Z Z J U C L U D T U N W G C Q S
(+ mod 26)

Many other encryption algorithms and methods followed the Vigenère Cipher. The list includes the Rotor Machine and the renowned Enigma Machine, both largely influenced by the industrial age. Consequently, most of the encryption algorithms that followed the Vigenère Cipher were fundamentally similar to the substitution cipher but more complex structurally. [3]

3

Modern Encryption

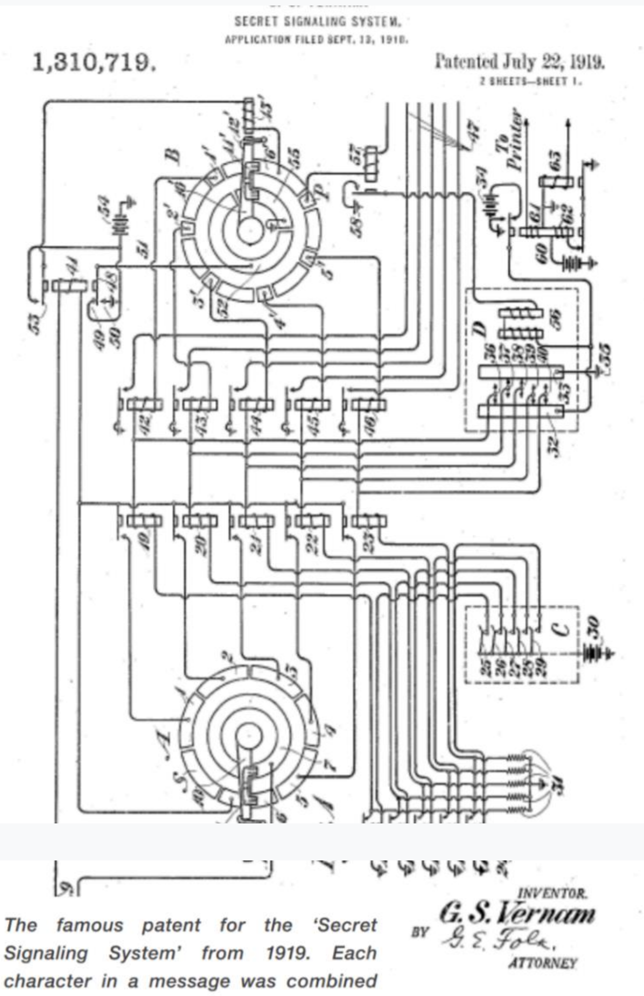
"There are two types of encryption: one that will prevent your sister from reading your diary and one that will prevent your government".

-Bruce Schneier

`

Gilbert Vernam

Joseph Mauborgne



[public domain]

The famous patent for the 'Secret Signaling System' from 1919. Each character in a message was combined with a character on a paper tape key.

3. 1. One Time-Pad

In 1918, Gilbert Vernam and Joseph Mauborgne developed an encryption algorithm called *One-Time Pad* which even today is still unbreakable for cipher-text-only-attack. In other words, with no information about the plain text, it is impossible to break the cipher. The One Time Pad is applied when first the entire message is represented in binary. The message can be an image, an audio or a text. The key is a ***random*** sequence of 0s and 1s of the same length as the message. Once the key is used, it can never be used again. The encryption process consists of adding the key to the message mod 2, bit by bit. This process is called *exclusive or* and denoted by XOR ($\oplus$). Notice that the only time the sum of two binary numbers is 1 is when one number is 0 and the other is 1. Hence, the addition is performed as follow: $0 \oplus 0=0$, $1 \oplus 0=1$, $0 \oplus 1=1$, $1 \oplus 1=0$. To look at an example:

Plaintext:	01001111
Key:	01010110
Cipher text:	00011001

The One-Time Pad is a **symmetric key algorithm**, which means the encryption key and decryption key are the same[3, pp 40]. To decrypt, add the key to the cipher text: 00011001 + 01010110= 01001111. As one may notice, *the exclusive or* has the nice property of being reversible, which makes encryption and decryption smooth.

3. 2 A Mathematical Analysis of The One-Time Pad

In this section we introduce a theoretical approach to the security of cryptosystems. The basic question is the following: If the eavesdropper, let's call her Eve, observes a piece of ciphertext, does she gain any knowledge about the encryption key that she did not already have? Claude Shannon, an American mathematician and cryptographer known as "the father of information theory", in his paper *Communication Theory of Secrecy System* [6], answered this question in great detail. We will focus on the ramifications of Shannon's proof on the One Time Pad. We will prove using mathematics why the *One-Time Pad* cipher is unbreakable with *cipher-only-attack*. Before doing that, we will define a cipher mathematically, and describe what it means for a cipher to have **perfect secrecy**.

A cipher c defined over a key space $\mathcal{K}$, a message space $\mathcal{M}$, and a cipher space C, is a pair of *efficient*[d] algorithms (E, D) such that for all $m \in \mathcal{M}$, and $k \in \mathcal{K}$, D (k, c) = m where the key k has the same length as the message.

c: E (k, m) = $k \oplus m$

D (k, c) = $k \oplus c$

$\mathcal{K} = \mathcal{M} = C = \{0,1\}^n$ = n-bits strings

Consistency Test

The definition of a cipher requires it to be decipherable; that is, to return the plaintext message

[d] Efficient here means runs in polynomial time. Practically speaking, it means runs in a certain time period, and not indefinitely.

when the correct key is presented. A cipher that cannot be deciphered given the correct key is not consistent. With the One-Time pad, the consistency test works perfectly.

Example:

$D(k, E(k, m)) = D(k, k \oplus m) = k \oplus (k \oplus m)$

$= (k \oplus k) \oplus m$ Associative law

$= 0 \oplus m$ Inverse law

$= m$ Identity law QED [6, pp 6]

Perfect Secrecy

A cipher *c* defined over *(K, $\mathscr{M}$, C)* has *perfect secrecy* if for all m_o, $m_1 \in \mathscr{M}$ where the length of m_o equals the length of m_1, and for all $c \in C$: $\Pr[E(k, m_o) = c] = \Pr[E(k, m_1) = c]$ where k is uniform [e] in *K,* and Pr represents the probability to encrypt a message m with a key k.

In other words, the ciphertext yields no information about the plaintext that was not already known.

We shall show that The One-Time Pad has perfect secrecy

The proof will consist of showing that the probability to encrypt any messages using a key k will stay the same (constant).

Proof: For all $m \in \mathscr{M}$: $\Pr[E(k, m) = c] = \dfrac{\#\,\textit{Keys such that}\ E(k,m)=c}{|K|}$

For the One-Time Pad $E(k, m) = c$ implies that $k \oplus m = c$ (definition of the One-Time cipher)

[e] A uniform distribution on a finite set of numbers is one in which each possible number is equally probable. (See The Art of Computer Programming Third Edition pp-2).

=> $k=m \oplus c$ (cancelation law holds since XOR is an abelian group in $\{0,1\}^n$)

=> $\#\{k \in \mathcal{K}: E(k, m) = c\} = 1$. For all m, c.

Therefore, $\Pr[E(k, m) = c] = \frac{\#\,\mathit{Keys\ such\ that}\ E(k,m) = c}{|K|} = \frac{1}{|K|} = \textbf{constant}$

QED. Thus, The One-Time Pad has perfect secrecy. [6]

4

Pseudo Random Bit Generator (PRG)

"Anyone who considers arithmetical methods for producing random digits is, of course, in a state of sin". *-John Von Neuman (1951)*

Randomness plays a key part in Cryptography. The *One-Time Pad*, which is the only cipher proven mathematically to be secure [6], uses randomness at its core. However, the One-Time Pad is not practical since it requires the key to be the same length as the message. In effect, with the One-Time Pad, the problem is changed to how to transmit the key securely. This is where the Pseudo Random Bit Generator (PRG) enters the scene. The PRG furthers the idea behind the One-Time Pad by making it practical. To do that, it replaces the random key with something that looks like a random key called a *pseudo random key*. Before exploring how PRG functions are obtained, we first describe the concept behind what different PRG functions are trying to accomplish: generating **random numbers**. Randomness is not a trivial concept to define nor to reproduce. To quote Donald Knuth, a prominent computer scientist and one of the first figures to mathematically formalize Computer theory, *People who think about this topic almost invariably get into philosophical discussions about what the word random means. In a sense, there is no such thing as a random number; for example, is 2 a random number? Rather, we speak of a* ***sequence of independent random numbers*** *with a specified distribution, and this means loosely that each number was obtained merely by chance, having nothing to do with other numbers of the sequence, and that each number has a specified probability of falling in any given range of values*. [7, pp. 2]

Knuth goes on later in the book and gives a “quantitative definition” of random behaviors, and random sequences using the axioms of probability.

One of the most used PRG functions is the **linear congruential generator**. Many high-level languages including C and Java use it. A linear congruential generator produces a sequence of number $x_o, x_1, x_2, \ldots,$ where

$$X_n = aX_{n-1}+b \pmod m$$

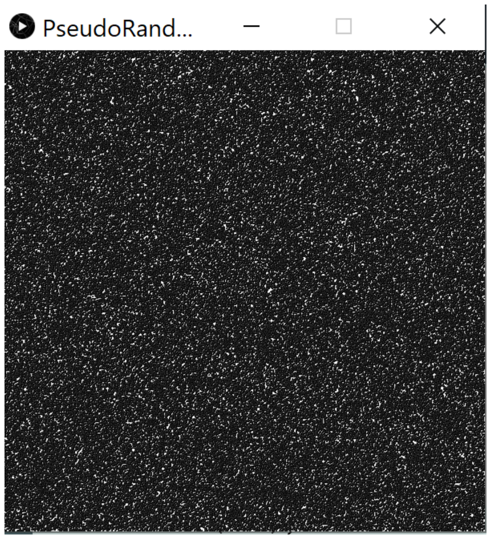
The number x_o is the seed, and the numbers a, b and m are parameters that govern the relationship. A linear congruential generator is very easy to compute but also has its limitations. One of the main problems with it is that after a certain point, it starts repeating; in other words, it has a period. One way to partially solve the problem is to use a linear congruential generator with an extremely large period. [f]

Pseudo Random Bit Generators take a seed, an s-bit string, and extends it to an n-bit string, where the seed *s* is very small compared to *n.* This is where PRGs mainly differ with the One-Time Pad. While the One-Time Pad has a key that is randomly generated using simulation

[f] Donald Knuth, in his book The Art of Computer Programming Vol 2, does an extensive analysis on the theory of Random numbers. On page 184 (Third Edition), he gives a summary of what constitutes a good linear congruential generator.

processes such as the thermal noise from semiconductor resistors or throwing a die, PRGs take a seed and expands it using deterministic processes. At the same time, not all PRGs are created equal; some are more efficient than others. One of the most popular secure pseudo-random number generators is the **Blum-Blum-Shub (BBS).** To understand why the BBS is more secure than other PRGs requires a profound understanding of number theory and probability [8]. One intuitive way to see how efficient a PRG function is can be done by observing patterns of numbers generated by the PRG.

To see an example, look at figure 4.1. The code is written so that the numbers generated by the *congruential generator* is displayed in a 2D black background screen as white dots [VI. 3]. As we can see from figure 4.1 (a), no real pattern emerges compared to, say, the Middle Square Method[g] for generating random numbers (Figure 4.1.b).



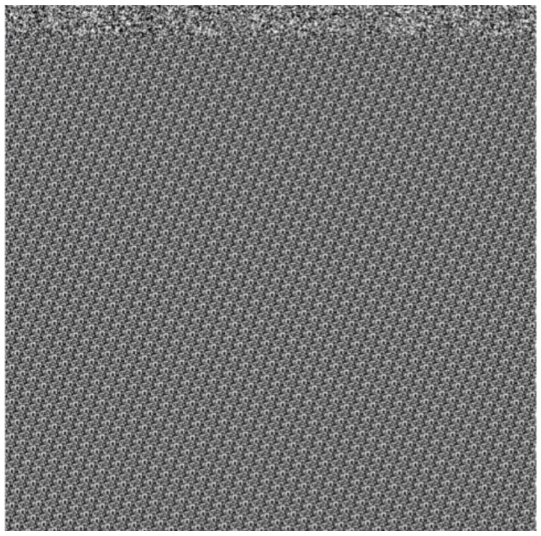


Figure 4.1 (a) Figure 4.1 (b)

[g] The Middle Square method is a method of generating pseudo random numbers. In practice, it is not good since it has a very short period. The method was invented by John van Neuman to illustrate the point that, “Anyone who considers arithmetical methods for producing random digits is, of course, in a state of sin”.

5

Elliptic curve Cryptography

"There is geometry in the humming of strings."

-Pythagoras

In 1985, Neal Koblitz and Victor S. Miller suggested independently the use of elliptic curves in cryptographic schemes. It was not until 2004 that elliptic curves became widely used. One of the advantages of elliptic curve schemes in cryptography is that they seem to offer a level of security similar to classical cryptosystems that use much larger key sizes. Consequently, elliptic curve schemes are much faster than classical cryptosystems.

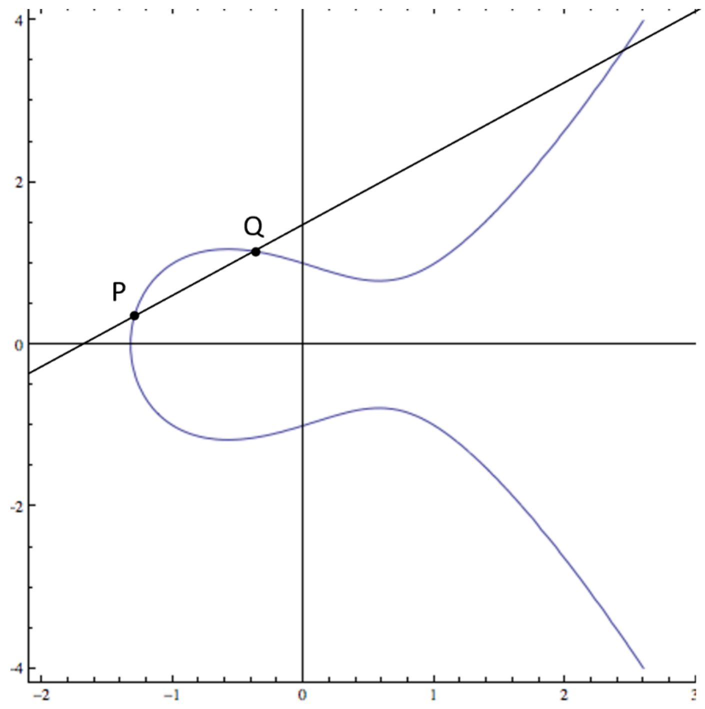
An elliptic curve *E* is the graph of an equation $E: y^2 = x^3 + ax + b$ where a and b are in whatever is the appropriate set (rational numbers, real numbers, integer mod p, etc.). There is a requirement that the discriminant $\Delta = 4a^3 + 27b^2$ be nonzero[h]. For reasons to be explained later, we introduced an extra point $\mathcal{O}$, that is "***at infinity***", so E is the set

$$E=\{(x,y): y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$$

Elliptic curves are not ellipses. They received their name from their relation to elliptic integrals such as $\int_s^t \frac{dx}{\sqrt{x^3+ax+b}}$ and $\int_s^t \frac{xdx}{\sqrt{x^3+ax+b}}$ that arise in the computation of the arc length of ellipses. We can use geometry to make the points of an elliptic curve into a **Group** [12, pp. 304].

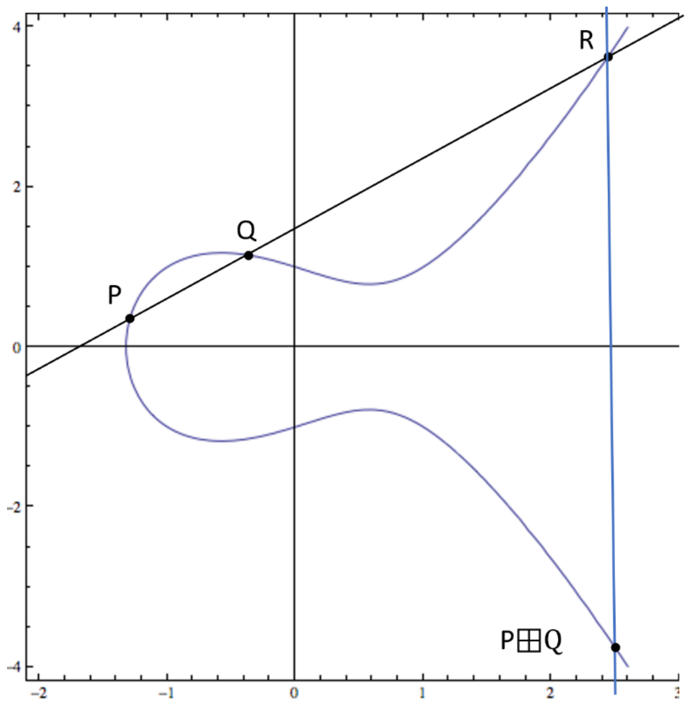
5.1 The Geometry of Elliptic Curves

Adding points on an Elliptic Curve E



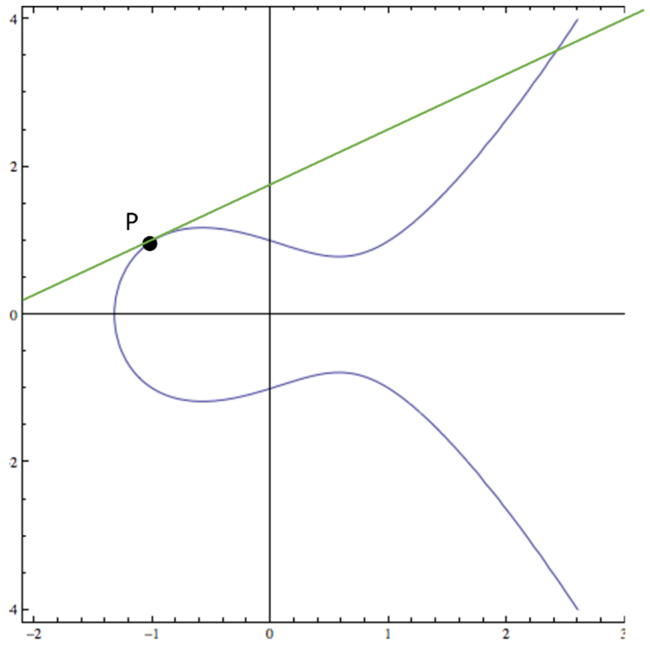
Start with point P and Q on *E*, and draw a line L through the point P and Q.

[h] The above condition ensures that the elliptic curve is nonsingular. If an elliptic curve is singular, the group structure on it is isomorphic to the multiplicative group of a quadratic extension of a field. Consequently, the discrete logarithm problem is easy to solve in such cases. Hence, there is no reason to use the curve.

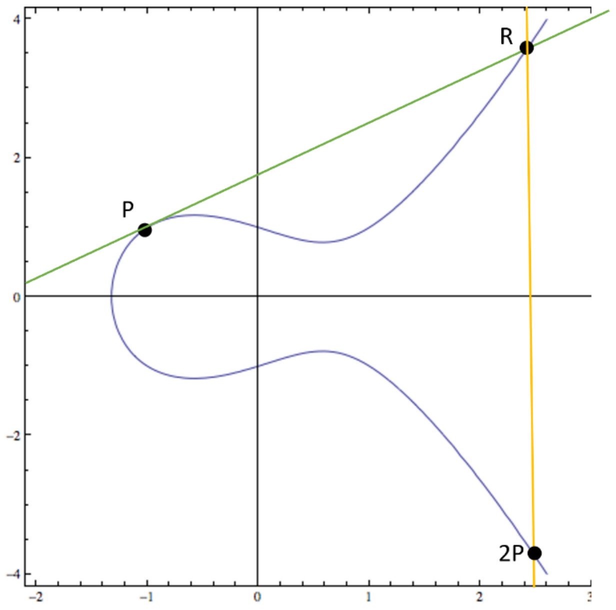


The line L intersects the curve in a third point. Call the third point R, and draw a vertical line through R. It intersects the curve in another point. ***We define the sum of P and Q on E to be the reflected point of R.*** We denote it by P⊞Q or simply by P+Q.

Adding a point to itself on an Elliptic Curve

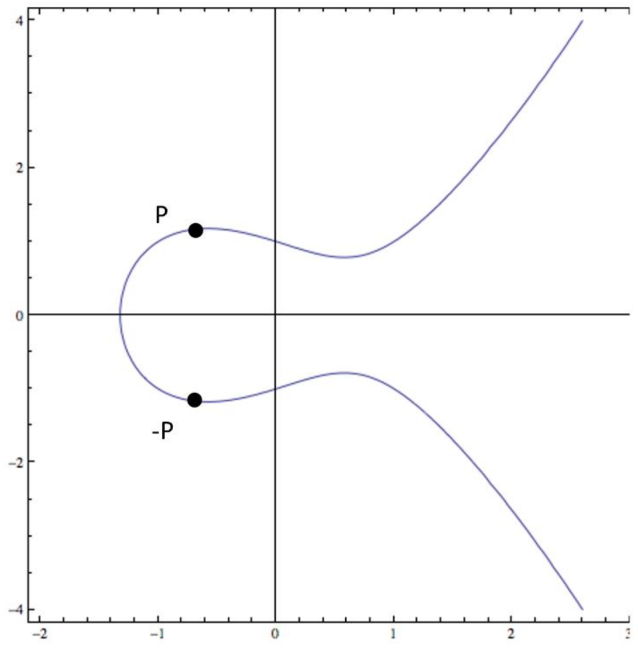


How do we add a point P to itself since there are infinitely many lines that pass through it? If we think of adding P to Q and let Q approach P, then the line L becomes the tangent line to *E* at P.

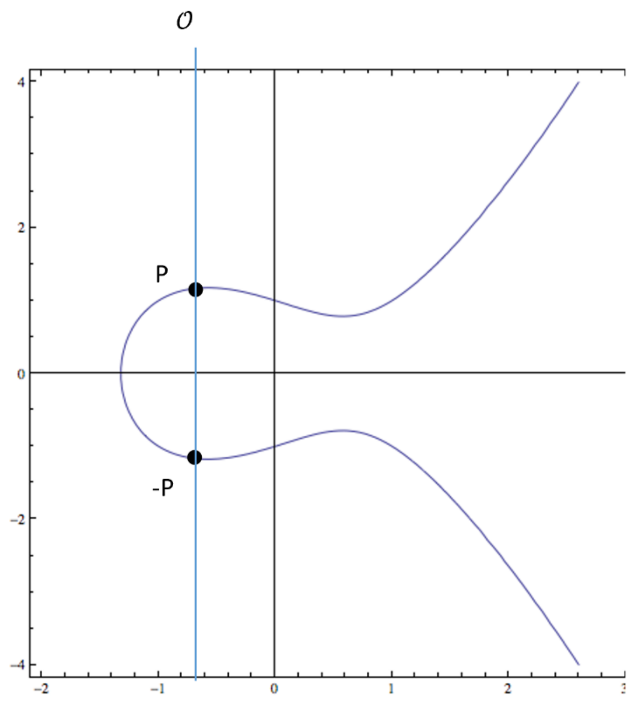


The tangent line L intersects the curve at R, reflect across the x-axis and call the resulting point **P⊞P= 2P.**

Adding a point to its reflection on the x-axis and the Extra point "At infinity"



Let P be a point on the curve E. We denote the reflected point by -P. We have a problem; the vertical line L through P and -P does not intersect the curve anywhere.



Since there is no point in the plane that works, we create an extra point $\mathcal{O}$ “at infinity”.

Rule: $\mathcal{O}$ is the point on every vertical line

5.2 The Algebra of Elliptic Curves

Properties of ***Addition*** on *E*

Theorem: The addition law on *E* has the following properties:

(a) $\mathbf{P} + \mathcal{O} = \mathcal{O} + \mathbf{P} = \mathbf{P}$ for all P $\in E$

(b) $\mathbf{P} + (-\mathbf{P}) = \mathcal{O}$ for all P $\in E$

(c) $\mathbf{P} + (\mathbf{Q} + \mathbf{R}) = (\mathbf{P} + \mathbf{Q}) + \mathbf{R}$ for all P, $Q, R \in E$

(d) $\mathbf{P} + \mathbf{Q} = \mathbf{Q} + \mathbf{P}$ for all P, $Q \in E$

In other words, the addition law **+** makes the points on *E* into a commutative group. All of the group properties are trivial to verify except for the associative property (c).

An interesting feature of addition on E is that it makes sense to define multiples of a point P as follows:

$0P = \mathcal{O}$

$1P = P$

$2P = P + P$

$3P = P + P + P$

$4P = P + P + P + P$

$\ldots = \ldots$ [12, pp. 303]

Let's find the formulas for P+Q. Suppose P(x_1,y_1) and Q(x_2,y_2). The line L through P and Q is given by the equation $y= mx+b$, assuming that $P \neq Q$ we have:

$m=\frac{y2-y1}{x2-x1}$ and $b=y_1 - mx_1$

If P=Q, we can take the derivative at (x_1,y_1) to find the slope:

$d(y^2)= d(x^3+ax+b)$

$2ydy = (3x^2+ax)\, dx$

$\frac{dy}{dx}=\frac{3x^2+ax}{2y}$ so if P= Q at (x_1,y_1) we obtain $m= (3x_1^2+a)/2y_1$ and $b=y_1 - mx_1$

Now that we have the equation of the line, let find the third point (x_3,y_3) which is the intersection of the line $y= mx+b$ and the curve E: $y^2= x^3 + ax + b$:

$(mx+b)^2=x^3 + ax + b$ Since we know that (x_1,y_1), (x_2,y_2) and (x_3,y_3) are all solutions, we know that

$0= x^3 + ax + b - (mx+b)^2$

$0= (x-x_1)(x-x_2)(x-x_3)$

$0= x^3 - (x_1+ x_2+ x_3)x^2 + (x_1x_2+ x_2x_3+ x_3x_1)x - x_1x_2x_3$

By matching coefficients, we can conclude that $m^2= (x_1+ x_2+ x_3)$ so we can conclude that $x_3= m^2- x_1- x_2$ and therefore $y_3 = mx_3+b$. Hence, P+Q=$(x_3,-y_3)$

In Conclusion:

If P≠Q and $x_1= x_2$: P+Q= $\mathcal{O}$

If P = Q and $y_1= y_2=0$: P+Q= $\mathcal{O}$

Otherwise: P+Q = $(\boldsymbol{m^2- x_1- x_2, -m^3+m(x_1+ x_2) -b})$ [12, pp. 299]

5.3 The Group of Points on E with coordinates in a finite field K

Theorem: (Poincare≈ 1900)

Let K be a field and suppose that an elliptic curve E is given by an equation of the form $E : y^2 = x^3 + ax + b$ with $a, b \in$ K. Let E(K) denote the set of points of E with coordinates in K, E(K) = $E=\{(x,y): y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$. Then E(K) is a subgroup of the group of all points of E.

From this theorem, we can conclude that the formulas giving the **group law** on E are also valid when the points have coordinates in any field. The field most used in cryptography is the finite field of integers modulo p where p is a prime number. For example, we can take the points in $\mathbf{F}_p$.

E: $y^2 = x^3 - 5x + 8 \ (mod\ 37)$ contains the points (this example was taken from [9, pp. 28]):

P(6,3) ∈ $E(\mathbf{F}_{37})$ and Q=(9,10) ∈ $E(\mathbf{F}_{37})$. Using the addition formulas, we can compute in $E(\mathbf{F}_{37})$

2P=(35,11), 3P=(34,25), 4P=(8,6), 5P=(16,19), …$x^3 - 5x + 8$ (*mod 37*)

P+Q=(11,10)…. , 3P+4Q=(31,28),…

Substituting in each possible value x = 0,1,2…,36 and verifying that if $x^3 - 5x + 8$ is a square modulo 37, we deduce that $E(\mathbf{F}_{37})$ consists of 45 points modulo 37:

$(1, \pm2)$, $(5, \pm21)$, $(6, \pm3)$, $(8, \pm6)$, $(9, \pm27)$, $(10, \pm25)$, $(11, \pm27)$, $(12, \pm23)$, $(16, \pm19)$, $(17, \pm27)$, $(19, \pm1)$, $(20, \pm8)$, $(21, \pm5)$, $(22, \pm1)$, $(26, \pm8)$, $(28, \pm8)$, $(30, \pm25)$, $(31, \pm9)$, $(33, \pm1)$, $(34, 25)$, $(35, \pm26)$, $(36, \pm7)$, $\mathcal{O}$

5.4 The Discrete Logarithm Problem

Fix a group G and an element g ∈ G. The discrete logarithm problem (DLP) for G is:

Given an element h in the subgroup generated by g, find an integer m satisfying $h=g^m$

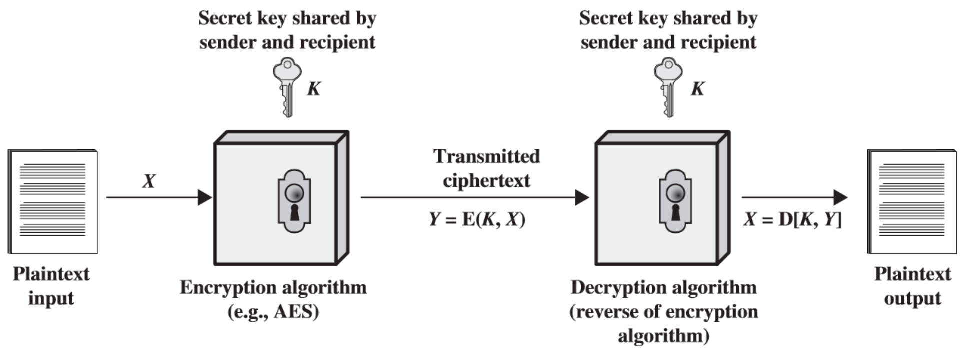
The smallest integer m satisfying $h=g^m$ is called the logarithm (or index) of h with respect to g, and is denoted $m=log_g(h)$ or $m=ind_g(h)$

The discrete logarithm problem is used as the fundamental hard problem in many cryptographic schemes including public key cryptography, encryption, digital signatures, and hash functions [3, pp. 201].

Diffie Hellman Key-exchange (DHK)

In beginning chapters, we have talked about some encryption methods. However, encryption can only happen if there is a key that both parties shared (See Figure 5.4). It is the key that enables both parties to encrypt and decrypt their message. How parties establish the key in an encryption model is assumed to be a given. Both parties may have met in secret and shared the key or done it some different ways.

Figure 5.4


Secret key shared by sender and recipient
K
Secret key shared by sender and recipient
K
X
Transmitted ciphertext
Y = E(K, X)
X = D[K, Y]
Plaintext input
Encryption algorithm (e.g., AES)
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

As Cryptography became more and more popular and useful, a question surfaced: how can two parties share a key in an unprotected channel? It took three graduate students from Stanford University in the 1970's, Ralph Merkle, Whitfield Diffie, and Martin Hellman, to design such a protocol (Figure 5.4.1). [10]

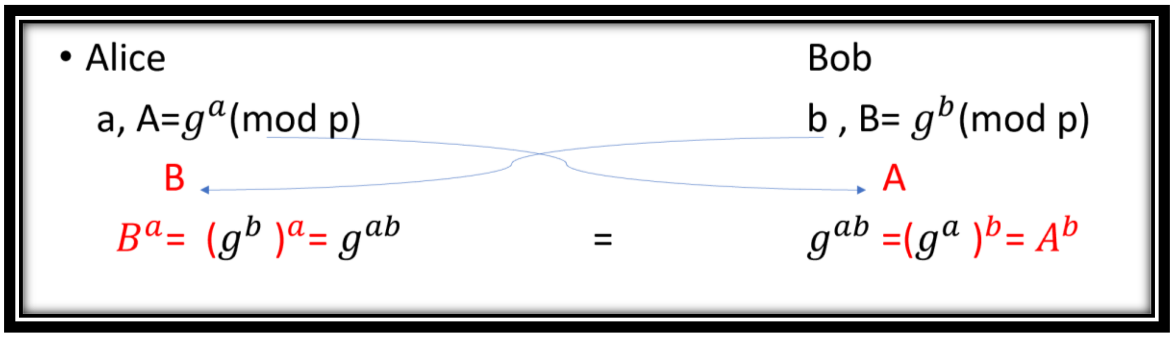
Here is how Alice and Bob establish a private key. The communication is done over a public channel.

1. Either Alice or Bob selects a large prime p and a primitive root g[i] mod p. Both p and g are made public.

[i] G is a primitive root mod p if for every if for every integer a coprime to n, there is an integer k such that $g^k \equiv a$ (mod n). G is called **generator** since it can generate the all set, that is, every positive integer including zero less than p.

2. Alice chooses a secret random number $\boldsymbol{a}$ with $1 \leq a \leq p-2$, and Bob does the same, selects a random number $\boldsymbol{b}$ with $1 \leq b \leq p-2$.
3. Alice sends A= g^a (mod p) to Bob and Bob sends B= g^b (mod p) to Alice.
4. Using the messages each one received, they can compute the private key K. Alice calculates K by K=($g^b)^a$ (mod p) and Bob calculates K by K=($g^a)^b$

Figure 5.4.1



The problem for the eavesdropper given p, g , A and B, is then can she compute $\boldsymbol{g^{ab}}$ **(mod p)**? Computing $\boldsymbol{g^{ab}}$ given $\boldsymbol{g^a}$ and $\boldsymbol{g^b}$ requires solving the discrete logarithm problem. The difficulty of the discrete logarithm problem varies depending on the group. For some groups, DLP is very easy. This includes $\mathbb{Z}/m\mathbb{Z}$ under addition (Euclidean algorithm), $\mathbb{R}^*$ under multiplication (analytic logarithm),etc. Groups where the DLP is difficult includes $\mathbb{F}^*{}_p$ under multiplication.

The best-known algorithm for solving the DLP in $\mathbb{F}^*{}_p$ takes time

$O(e^{c\sqrt[3]{(\log p)(\log\log p^2}})$. The time complexity is not completely exponential because of the cube root at the exponent. This is called **subexponential** since it is faster than exponential time complexity, but slower than polynomial time complexity. For cryptographic purposes, it would be better to use a group G for which solving the DLP has a fully exponential time complexity[j]. This brings us to Elliptic curve Diffie Hellman Key-exchange.

5.5 Elliptic Curve Diffie Hellman Key-exchange (ECDHK)

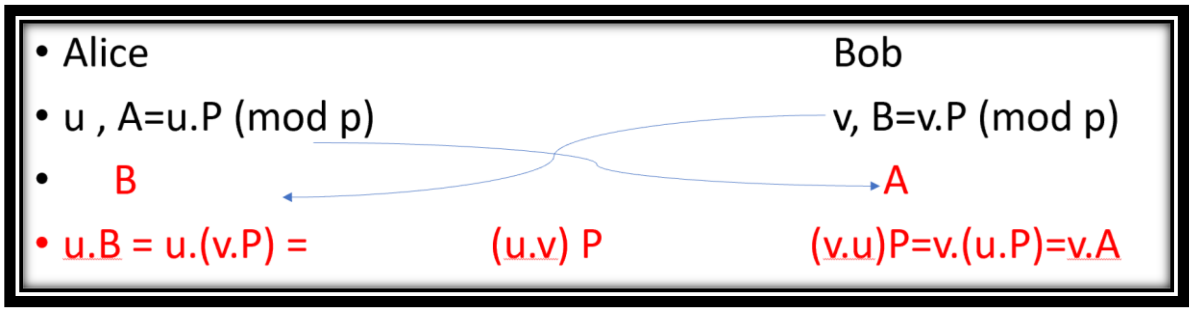
The same protocol of the Diffie Hellman key-exchange is used but with a minor change. The only difference is that we are using the protocol under a different group (the group we just have constructed on the Elliptic Curve) since the discrete logarithm problem is harder there. So instead of computing g^a or g^b (mod p), we now compute u.P and v.P instead. The steps for the DHK under the Elliptic curve group we have constructed (5.2, 5.3) then becomes:

1. Either Alice or Bob selects a large prime p, a curve E: $y^2 = x^3 + ax + b \ (mod\ p)$, and a point P on the curve. Everything is public at this point.
2. Alice chooses a secret random number u with $1 \leq u \leq p-2$, and Bob chooses also a random number v with $1 \leq v \leq p-2$.
3. Alice sends A= u.P to Bob and Bob sends B= v.P to Alice.
4. Using the messages that each one has received, they can compute the private key K. Alice calculates K by K= u.B=u.(v.P) (mod p) and Bob calculates K by K= v.A= v(u.P) (mod p).

One can note that associativity is crucial for the DHK protocol to work. That is why the operation under the elliptic curve was built with a group structure as an end goal (See figure 5.5).

[j] This will mean computationally infeasible to break at any reasonable time.

Figure 5.5



The eavesdropper can see the point P, the prime number p, the point A=u.P(mod p) , B=v.P (mod p) and the curve E. Can she compute ***u.v.P*** ? It turns out the best-known algorithm to compute ***u.v.P*** for n-digit prime is ***exponential***. Hence, we have the same security with smaller primes[k][12, pp. 316].

5.6 Elliptic Curve Diffie Hellman Key-exchange (ECDHK) Using Java

The code below allows two people, Alice and Bob, to complete the ECDHK protocol. Alice secretly chooses u and Bob chooses v. As we know from the protocol described in (5.5), Alice computes A=u.P and Bob computes B = v.P . Using the crypto java library and the SunEC implementation of the ECDHK protocol, Alice and Bob computes K=u.B= v.A= u.v.P which is *the private key* . [11]

[k] In practice 77 digits (256 bits) primes are used (10 times faster than DHK mod p).

```
import java.math.BigInteger;

import java.security.InvalidAlgorithmParameterException;

import java.security.InvalidKeyException;

import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.NoSuchAlgorithmException;

import java.security.NoSuchProviderException;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.security.spec.ECGenParameterSpec;

import javax.crypto.KeyAgreement;

/**

 * This code is the implementation of the Elliptic Curve Diffie
Hellman key-exchange. To implement it we used the java library crypto
, and the curve secp192k1.

 * @author Edmond Mbadu

 */

public class ECCPrivateKey {
```

```
public static void main(String[] args) throws

NoSuchAlgorithmException, NoSuchProviderException,

InvalidAlgorithmParameterException, InvalidKeyException {

KeyPairGenerator keyPairGen;

// Specify that we are using the elliptic curve implementation

// of the Diffie Hellman protocol

keyPairGen = KeyPairGenerator.getInstance("EC", "SunEC");

// Initialize the parameter class

ECGenParameterSpec Escp;

// Get the parameters of the particular curve secp192r1

Escp = new ECGenParameterSpec("secp192k1");

// Initialize the curve

keyPairGen.initialize(Escp);

// from Step 1 of the Diffie Hellman Protocol Generate the private key

// for Alice

// But first generate the Key pair:

// The Key pair contains both the private and the public key for Alice

KeyPair kpAlice = keyPairGen.generateKeyPair();
```

```
// Now generate first the private key for ALice

PrivateKey privKeyAlice = kpAlice.getPrivate();

// Get the public key for Alice

     PublicKey pubKeyAlice = kpAlice.getPublic();

// Display both the public key that will be sent to Bob

System.out.println("Alice: " + privKeyAlice.toString());

System.out.println("Alice: " + pubKeyAlice.toString());

// Repeat the same steps for Bob

KeyPair kpBob = keyPairGen.generateKeyPair();

// Now generate first the private key for Bob

PrivateKey privKeyBob = kpBob.getPrivate();

// Get the public key for Bob

PublicKey pubKeyBob = kpBob.getPublic();

System.out.println("Bob: " + privKeyBob.toString());

System.out.println("Bob: " + pubKeyBob.toString());

// This is step 4 of the protocol

KeyAgreement ecdhAlice=KeyAgreement.getInstance("ECDH");

// Initialize the private key of Alice
```

```
           ecdhAlice.init(privKeyAlice);

// Pass the value computed by Bob which is public

ecdhAlice.doPhase(pubKeyBob, true);

// Do the same thing with Bob

KeyAgreement ecdhBob=KeyAgreement.getInstance("ECDH");

ecdhBob.init(privKeyBob);

ecdhBob.doPhase(pubKeyAlice, true);

// The result( which is converted to Hex)  is the same for both
//Alice and Bob

System.out.println("Secret key computed by Alice: 0x"+(new
BigInteger(1,ecdhAlice.generateSecret()).toString(16).toUpperCase()));

System.out.println("Secret key computed by Bob  : 0x"+(new
BigInteger(1,ecdhBob.generateSecret()).toString(16).toUpperCase()));

     }

}
```

Below is a sample output for the program. Note that the output will be different each time the code is run, since the private key is randomly generated.

```
Alice: sun.security.ec.ECPrivateKeyImpl@2d35

Alice: Sun EC public key, 192 bits
```

```
  public x coord:
1678001189984502078598529146428480144626274321861390796091
  public y coord:
4394559801515060113724881501494424321697524808755576622058
  parameters: secp192k1 (1.3.132.0.31)
Bob: sun.security.ec.ECPrivateKeyImpl@ffffedb7
Bob: Sun EC public key, 192 bits
  public x coord:
1160546040565414615908723599086907275960949007297908563955
  public y coord:
3375881982021148141850614180142361441976212418917632951325
  parameters: secp192k1 (1.3.132.0.31)
```

Secret key computed by Alice:

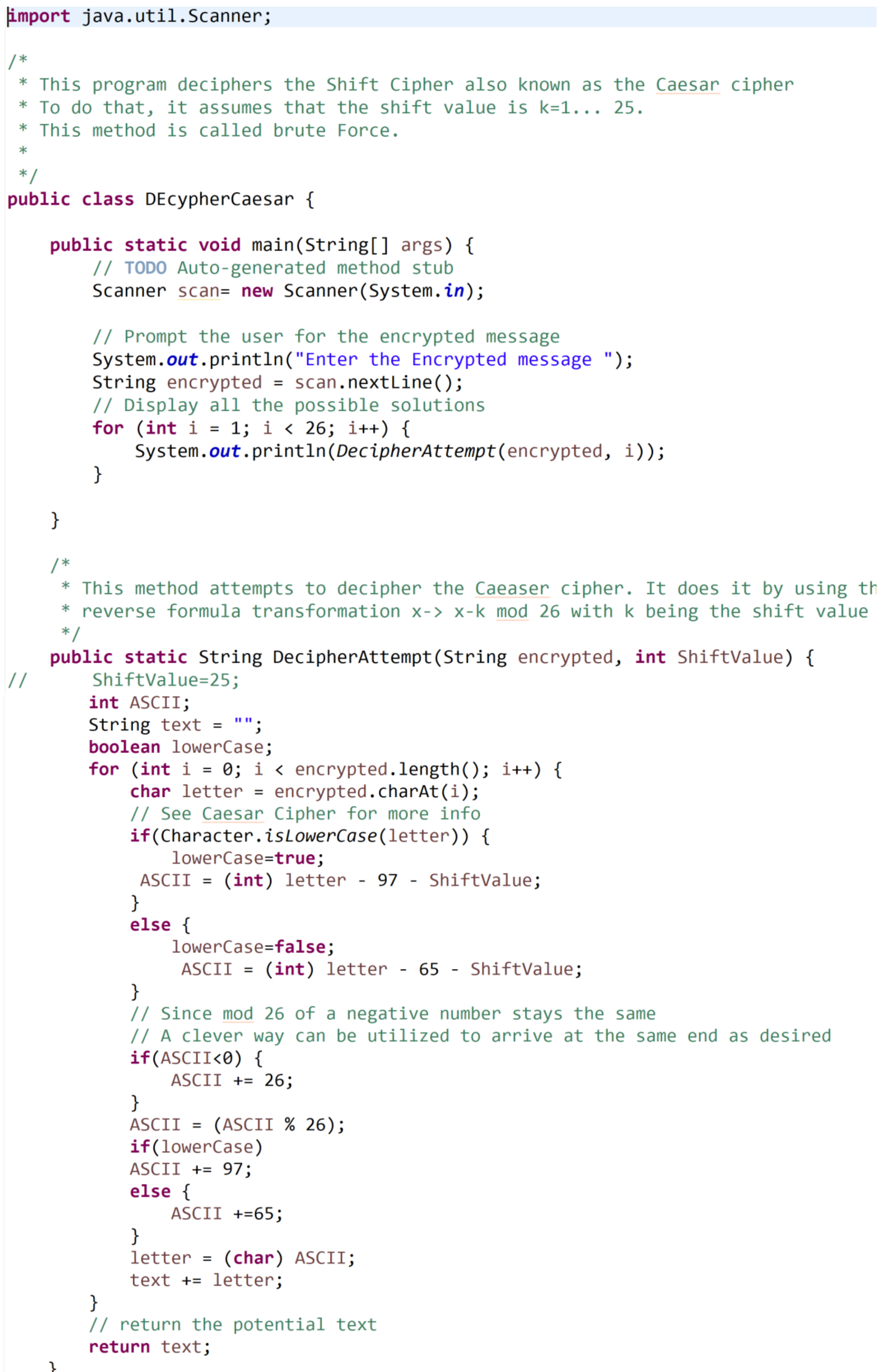
0xC2342402D40CCFED69F55E0E7F6D0CD16D70EAD007ABCC0E

Secret key computed by Bob :

0xC2342402D40CCFED69F55E0E7F6D0CD16D70EAD007ABCC0E

Source Code

1. Breaking the Caesar Cipher



```
import java.util.Scanner;

/*
 * This program deciphers the Shift Cipher also known as the Caesar cipher
 * To do that, it assumes that the shift value is k=1... 25.
 * This method is called brute Force.
 *
 */
public class DEcypherCaesar {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner scan= new Scanner(System.in);

        // Prompt the user for the encrypted message
        System.out.println("Enter the Encrypted message ");
        String encrypted = scan.nextLine();
        // Display all the possible solutions
        for (int i = 1; i < 26; i++) {
            System.out.println(DecipherAttempt(encrypted, i));
        }

    }

    /*
     * This method attempts to decipher the Caeaser cipher. It does it by using th
     * reverse formula transformation x-> x-k mod 26 with k being the shift value
     */
    public static String DecipherAttempt(String encrypted, int ShiftValue) {
//      ShiftValue=25;
        int ASCII;
        String text = "";
        boolean lowerCase;
        for (int i = 0; i < encrypted.length(); i++) {
            char letter = encrypted.charAt(i);
            // See Caesar Cipher for more info
            if(Character.isLowerCase(letter)) {
                lowerCase=true;
             ASCII = (int) letter - 97 - ShiftValue;
            }
            else {
                lowerCase=false;
                 ASCII = (int) letter - 65 - ShiftValue;
            }
            // Since mod 26 of a negative number stays the same
            // A clever way can be utilized to arrive at the same end as desired
            if(ASCII<0) {
                ASCII += 26;
            }
            ASCII = (ASCII % 26);
            if(lowerCase)
            ASCII += 97;
            else {
                ASCII +=65;
            }
            letter = (char) ASCII;
            text += letter;
        }
        // return the potential text
        return text;
    }
```

2. Blum-Blum-Shub

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

/**
 * This program is the implementation of the Blum-Blum-Shub algorithm (BBS)
 * pseudo-random bit generator, also known as the quadratic residue generator.
 * To do implement the algorithm: We set n = pq and choose a random integer x
 * that is relatively prime to n. To initialize the BBS generator, set the
 * initial seed to Xo= (X)^ 2 (mod n). The BBS generator produces a sequence of
 * random bits bi, • • • by 1. Xj = (mod n) 2. bj is the parity of the least
 * significant bit of Xj.
 *
 * Since we'll use very big numbers, we will use the BigInteger class
 * @author Edmond Mbadu
 *
 */


public class Blum_Blum_Shub {

  public static void main(String[] args) {

    BigInteger p = new BigInteger("24672462467892469787");
    BigInteger q = new BigInteger("39673689456783458980З");
    BigInteger n = p.multiply(q);


    BigInteger X = new BigInteger("87324564788847834901З");
    // Let find Xo
    BigInteger Xo = (X.multiply(X)).mod(n);

    // Generate the height first Xj

     // Set the initial value the same as Xo
     BigInteger Xj = (X.multiply(X)).mod(n);
     int i = 0;

     // Let generate the first 8 numbers
     String Bj = "";
     while (i < 8) {
       Xj = Xj.multiply(Xj).mod(n);
//       System.out.println(Xj);
       // Taking the parity of the least significant bit of each number,
       // and concatenates it to the string Bj
       Bj += Character.getNumericValue
           (Xj.toString().charAt(Xj.toString().length() - 1)) % 2;
       i++;

     }
     // The searched key is:
     System.out.println(Bj);


   }
```

3. Affine Cipher

```
import java.util.Scanner;

/*
 * Author: Edmond Mbadu
 * This program uses the Affine cipher ( an improvement on the substitution cipher)
 * to encrypt messages
 * Affine function F(x)= ax+b (
 * The affine cipher works if and only of the value of alpha is coprime to n
 * ( n= 26 in our case) .
 * This is because, if that is not the case, the function f(x) will not be one to one
 * making it possible for two ore more plain text letters to be encrypted to the same
 * cipher text letter.
 * In other words, it will be impossible to decrypt.
 *
 *
 */
public class AffineCipher {

  public static void main(String[] args) {

    long start = System.nanoTime();
    // Create an instance of the scanner class
    Scanner scan = new Scanner(System.in);
    //
    // Prompt the user for the message to be encrypted
    System.out.println("Enter the message: ");
    String message = scan.nextLine();

    // Prompt the user for the value of alpha, and beta
    System.out
      .println("Enter the value of alpha and beta  in that order separated by a space: ");
    String EncryptionValues = scan.nextLine();
    // Split the string that holds the values of info that
    //    will help with the encryption process

     String[] encrypt = EncryptionValues.split(" ");
     // Get the value of alpha
     int alpha = Integer.parseInt(encrypt[0]);
     int beta = Integer.parseInt(encrypt[1]);
     //    int shift = Integer.parseInt(encrypt[2]);
     // Display the encrypted message
     System.out.println(EncryptMessage(message, alpha, beta));
   }

   //
   public static String EncryptMessage(String message, int alpha, int beta) {
     // First check if alpha and 26 are coprime, beta is between 0 and 26
     if (isRelativelyPrime(alpha, 26) && beta < 26 && beta >= 0) {
       // Initialize values that will hold the encrypted letter and message
       char encryptedLetter;
       String encryptedMessage = "";
       // Encrypt the message
       for (int i = 0; i < message.length(); i++) {
         // If the current letter is lower case
         if (Character.isLowerCase(message.charAt(i))) {
           // Normalize i by reducing it to a value less than 26 ( letter) to apply
           // the transformation
           int normalize = ((int) message.charAt(i)) - 97;
           // Apply the affine transformation
           int F = (alpha * normalize + beta) % 26;
           encryptedLetter = (char) (F + 97);
         } else if (Character.isUpperCase(message.charAt(i))) {
           // Normalize it by reducing it to a value less than 26 ( letter) to apply
           // the transformation
           int normalize = ((int) message.charAt(i)) - 65;
           // Apply the affine transformation
           int F = (alpha * normalize + beta) % 26;
           encryptedLetter = (char) (F + 65);
      }
      // if the character is not a letter, do nothing
      else {
        continue;
      }
      // Append the current encrypted letter to the encrypted message
      encryptedMessage+=encryptedLetter;
    }
    // return the encrypted message
    return encryptedMessage;
  }
  // If the parameters are not good, return a message saying so
  return "Bad parameters. Try next time. ";
}

/**
 * This method tests if the two numbers are relatively prime. It uses the
 * Euclidean algorithm as the underlying structure
 *
 * @param a
 * @param b
 * @return
 */
public static boolean isRelativelyPrime(int a, int b) {
  if (gcd(a, b) == 1)
    return true;
  return false;
}

// Euclid algorithm
// Finds the greatest common divisor

  public static int gcd(int a, int b) {
    // Stopping case
    if (b == 0)
      return a;
    else
      return gcd(b, a % b);
  }
}
```

3. Linear Congruential Generator (Comment Needed)

```
  // Edmond Mbadu


  long a=1103515245L;
  int c=12345;
  long seed=1;
  long m=(long)Math.pow(2, 31);
  void setup() {
    size(600, 600);

    test();
  }

  void draw() {
    background(255);
    //stroke(0);
    for (int y=0; y<height; y++) {
      for (int x=0; x<600; x++) {
        if (nextRandFloat()>0.5) {
          fill(255);
          ellipse(x, y, 1, 1);
        }
      }
    }
  }

  long nextRand() {


    seed=(a*seed+c)%m;
    return seed;
  }


double nextRandFloat() {
  //
  return((double)nextRand()/m);
}
void test() {
  ArrayList<Long> results= new ArrayList<Long>();
  int i;
  for ( i=0; i<100000; i++) {
    long rand=nextRand();
    if (results.contains(rand)) {
      System.out.println(i);
      return;
    }

    results.add(rand);
  }

  System.out.println(i);
}
```

REFERENCES

[1] Source code of the Caesar cipher by the author.

[2] Singh Simon. The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography, London: Fourth Estate, p 78, ISBN 1-85702-879-1

[3] Trappe, Wade & Washington, Lawrence C. Introduction to Cryptography with Coding Theory. New-Jersey: Pearson Prentice Hall, 2006. Print.

[4] Stallings, Williams. Cryptography and Network Security with Principle and Practices. New-Jersey: Pearson Prentice Hall, 2011. Print.

[5] Schneier, Bruce. Applied Cryptography Protocols, Algorithms, and Source Code in C. Indianapolis: John Wiley & Sons Inc. 2015. Print.

[6] Shannon Claude. Communication Theory of Secrecy Systems. Bell System Technical Journal. 28 (4): 656-715. Doi: 10.1002/j.1538-7305.1949.tb00928.x.

[7] Knuth Donald (1998) “Chapter 3- Random numbers”. The Art of Computer Programming. Vol 2: Semi numerical algorithms.

[8] Blum, Lenore; Blum, Manuel; Shub, Mike (May 1, 1986). A simple Unpredictable Pseudo-Random Number Generator. SIAM Journal on Computing. 15 (2): 364-383. Doi: 10. 1137/0215025

[9] Silverman H, Joseph. An Introduction to the Theory of Elliptic Curves. Brown University and NTRU Cryptosystems, Inc. 2006.

[10]. Martin E. Hellman, Bailey W. Diffie, Ralph C. Merkle. Cryptographic Apparatus and Methods. United States Patent 4200770. April. 29, 1980.

[11] American National Standards Institute. Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). ANSI X9.62, 1998

[12] Hoffstein Jeffrey, Pipher Jill, Jeffrey Hoffstein. An Introduction to Mathematical Cryptography. New-York: Springer, 2014. Print.