**INDEX**

|  |  |  |
| --- | --- | --- |
| **Sl No.** | **Topic** | **Page No.** |
| I. | Introduction to Cryptography  I.A) Basic idea of Cryptography  I.B) Need for Cryptography  I.C) Principle of Cryptography  I.D) Types of Cryptography | 01  02  02-04  04-05 |
| II. | RSA  II.A) What is RSA?  II.B) RSA in data encryption  II.C) Steps in RSA algorithm  II.D) Robbin Miller Primality test  II.E) Advantages of RSA  II.F) Analysis of RSA | 06  06-07  07-10  10-11  11  11-12 |
| III. | ECC (Elliptic Curve Cryptography)  III.A) What is ECC?  III.B) Functions and algorithms used in ECC  III.C) Elliptic curve Diffie-Hellman algorithm (ECDH)  III.D) Elliptic curve digital signature algorithm | 13-14  15-16  16-18  18-19 |
| IV. | RSA vs ECC  IV.A) Why ECC is Better than RSA  IV.B) Different use cases of ECC and RSA  IV.C) Key strength of ECC and RSA | 20  21  21 |
| V. | Annexure  V.A) RSA code in java  V.B) ECC code in python | 22-26  27-40 |
| VI. | Basics of IBC | 28-36 |

**(I) INTRODUCTION TO CRYPTOGRAPHY**

I(A).BASIC IDEA OF CRYPTOGRAPHY:

***Cryptography*** is the science and art of designing and using codes to secure information and communications. The word cryptography comes from the Greek words Krypto’s, meaning hidden, and graphene, meaning writing. Cryptography has a long and fascinating history, dating back to ancient times, when people used simple substitution and transposition ciphers to conceal their messages. Today, cryptography has evolved into a complex and sophisticated field that relies on mathematics, computer science, engineering and physics.

The *main goal of cryptography* is to ensure the confidentiality, integrity, authenticity and availability of data and users. Confidentiality means that only the intended recipients can access the data. Integrity means that the data is not altered or corrupted during transmission or storage. Authenticity means that the data comes from a verified source and has not been forged or impersonated. Availability means that the data is accessible and usable when needed.

To achieve these goals, cryptography uses two main processes: encryption and decryption. Encryption is the process of transforming plain text (the original data) into cipher text (the coded data) using a key (a secret parameter). Decryption is the reverse process of recovering plain text from cipher text using the same or a different key. There are different types of encryption and decryption methods, such as symmetric key, asymmetric key and hash functions.

Cryptography has many applications in various domains, such as computer security, digital currencies, web browsing, authentication and cryptocurrencies. Cryptography helps to protect data and users from unauthorized access, tampering, theft and fraud. Cryptography also enables new forms of communication, collaboration and commerce that are based on trust, privacy and security.

***In this document*,** we will explore the concepts, types, examples and challenges of cryptography in more detail considering the cryptography techniques like: ***RSA*** (Rivest, Shamir, Adleman), ***ECC*** (Elliptic Curve Cryptography) and ***IBC*** (Identity Based Cryptography). We will also discuss the performance analysis of these cryptographic techniques and compare these techniques with each other to ensure which will be more useful depending upon our need and requirement.

I(B). NEED FOR CRYPTOGRAPHY:

Some applications of cryptography are described below-

* End-to-end encryption: A type of cryptography that encrypts messages with different keys for the sender and the receiver. No one else can read the messages, even if they intercept them. Used by apps like WhatsApp, Signal and Telegram.
* Authentication: A type of cryptography that checks the identity of a user or a device before allowing access. It encrypts or hashes credentials, such as passwords, tokens or biometrics. Used by systems and services like email, online banking and VPN.
* Electronic signatures: A type of cryptography that lets users sign documents digitally. It uses different keys to generate and verify signatures. The signature proves who signed the document and if it was changed. Used by apps like DocuSign, Adobe Sign and HelloSign.
* Secure web browsing: A type of cryptography that protects web traffic from being spied on or altered. It uses TLS or SSL protocols, which encrypt data with the same or different keys. The protocol also checks the identity of the web server and the web browser. Used by websites like Google, Facebook and Amazon.
* Computer passwords: A type of cryptography that secures access to computer accounts or devices. It uses hash functions, which produce a unique output from any input. The password is hashed before storing or sending it, so it cannot be read or cracked. Used by systems and devices like Windows, macOS and Android.

I(C). PRINCIPLES OF CRYPTOGRAPHY:

The Building Principle of cryptography are described below:

**1. Confidentiality**: In the realm of cryptography, confidentiality is paramount. It ensures that information remains secret and accessible only to the intended sender and receiver. Any compromise in confidentiality occurs if unauthorized entities manage to intercept and access the encrypted message. For example, if sender A encrypts confidential information for receiver B, an attacker C intercepting and accessing this information constitutes a breach of confidentiality.

**2. Authentication:** Cryptographic authentication verifies the identity of users, systems, or entities involved in communication. This process often employs mechanisms like usernames and passwords to ensure that only authorized individuals with pre-registered identities can access sensitive information securely.

**3. Integrity:** The principle of integrity guarantees that the information received is accurate and unchanged during transmission. In cryptography, both system integrity and data integrity are crucial. System integrity ensures that a cryptographic system functions as intended without unauthorized manipulation, while data integrity ensures that information remains unaltered in storage and during transmission.

**4. Non-Repudiation:** Non-repudiation mechanisms in cryptography prevent the denial of sent messages. It ensures that a sender cannot later deny sending a message, providing a level of assurance and accountability in cryptographic communication.

**5. Access Control:** Access control in cryptography involves role management and rule management. Role management dictates who should have access to cryptographic data, while rule management determines the extent of that access. The displayed information is contingent on the access privileges of the individual accessing it.

**6. Availability:** The principle of availability in cryptography emphasizes that cryptographic resources should be consistently accessible to authorized parties. For cryptographic systems, ensuring information availability is crucial to fulfilling user requests effectively.

**7. Ethical and Legal Issues:** Ethical considerations in cryptography revolve around individuals' right to privacy, property concerns related to information ownership, organizational rights to collect information (accessibility), and the obligation to maintain information accuracy, authenticity, and fidelity within cryptographic practices. Adhering to ethical principles is essential in navigating the complexities of cryptographic systems in alignment with legal frameworks.

I(D). Types of Cryptography:

On the basis of encryption techniques on the data, the cryptographic techniques are divided in two parts-

1. **Symmetric key cryptography**: It is an encryption technique that uses one key to encrypt and decrypt messages. The key is shared by the sender and the receiver, and it must be kept secret from anyone else who might want to access the messages. Symmetric key cryptography is also called *secret key cryptography* or *private key cryptography*, because only the parties who know the secret key can communicate securely. Symmetric key cryptography is commonly used in banking and data storage applications to prevent fraud, identity theft and data breaches. These applications require fast and efficient encryption and decryption of large amounts of data, which symmetric key cryptography can provide. Some examples of symmetric key algorithms are AES, DES, RC4 and Blowfish. These algorithms use different methods to transform the plain text into cipher text and vice versa, using the same key.
2. **Asymmetric key cryptography:** it is also known as public-key cryptography, represents a revolutionary paradigm in securing digital communication. Unlike traditional symmetric key cryptography, which relies on a shared secret key between communicating parties, asymmetric key cryptography employs a pair of distinct but mathematically linked keys – a public key and a private key. The public key is openly shared and used for encryption, while the private key, known only to the key's owner, is employed for decryption. This duality enables secure and efficient communication across untrusted networks. The strength of asymmetric key cryptography lies in its ability to provide a secure method for key exchange, authentication, and digital signatures without necessitating a priori sharing of secret keys. This cryptographic approach underpins the security infrastructure of various technologies, including secure web browsing (HTTPS), digital signatures, and email encryption, ensuring the confidentiality and integrity of digital communications in an increasingly interconnected and data-centric world.

**(II) RSA ALGORITHM**

II(A). What Is the RSA Algorithm?

The RSA algorithm is a public-key signature algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adleman. Their paper was first published in 1977, and the algorithm uses logarithmic functions to keep the working complex enough to withstand brute force and streamlined enough to be fast post-deployment. The image below shows it verifies the digital signatures using RSA methodology.

RSA can also encrypt and decrypt general information to securely exchange data along with handling digital signature verification. The image above shows the entire procedure of the RSA algorithm. You will understand more about it in the next section.

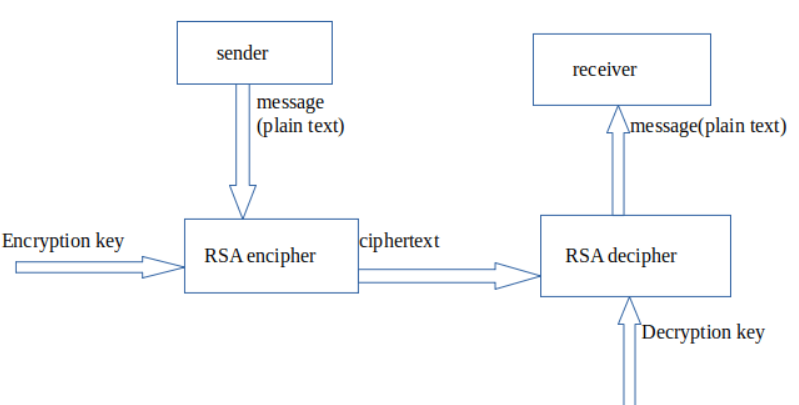


Figure – 2.1

II(B). RSA in Data Encryption:

When using RSA for encryption and decryption of general data, it reverses the key set usage. Unlike signature verification, it uses the receiver’s public key to encrypt the data, and it uses the receiver’s private key in decrypting the data. Thus, there is no need to exchange any keys in this scenario.

There are two broad components when it comes to RSA cryptography, they are:

1. *Key Generation:* Generating the keys to be used for encrypting and decrypting the data to be exchanged.
2. *Encryption/Decryption Function:* The steps that need to be run when scrambling and recovering the data.

We will now understand each of these steps in our next sub-topic.

II(C). Steps in RSA Algorithm:

A diagram of a key system

Description automatically generated

Figure – 2.2

**RSA algorithm uses the following procedure to generate public and private keys:**

1. Select two large prime numbers, p and **q**.
2. Multiply these numbers to find **n = p x q,** where **n** is called the modulus for encryption and decryption.
3. Choose a number **e** less than **n**, such that n is relatively prime to **(p - 1) x (q -1).** It means that **e** and **(p - 1) x (q - 1)** have no common factor except 1. Choose "e" such that 1<e < φ (n), e is prime to φ (n),  
   **gcd (e, d(n)) =1**
4. If **n = p x q,** then the public key is <e, n>. A plaintext message **m** is encrypted using public key <e, n>. To find ciphertext from the plain text, the following formula is used to get ciphertext C.  
   **C = me mod n**Here**, m** must be less than **n**. A larger message (>n) is treated as a concatenation of messages, each of which is encrypted separately.
5. To determine the private key, we use the following formula to calculate the d such that:  
   **De mod {(p - 1) x (q - 1)} = 1  
   Or  
   De mod φ (n) = 1**
6. The private key is <d, n>. A ciphertext message **c** is decrypted using the private key <d, n>. To calculate plain text **m** from the ciphertext c the following formula is used to get plain text m.  
   **m = cd mod n.**

**Example:**

This example shows how we can encrypt plaintext 9 using the RSA public-key encryption algorithm. This example uses prime numbers 7 and 11 to generate the public and private keys.

Explanation:

Step 1: Select two large prime numbers, p, and q.

p = 7

q = 11

Step 2: Multiply these numbers to find n = p x q, where n is called the modulus for encryption and decryption.

First, we calculate

n = p x q

n = 7 x 11

n = 77

Step 3: Choose a number e less than n, such that n is relatively prime to (p - 1) x (q -1). It means that e and (p - 1) x (q - 1) have no common factor except 1. Choose "e" such that 1<e < φ (n), e is prime to φ (n), gcd (e, d (n)) =1.

Second, we calculate

φ (n) = (p - 1) x (q-1)

φ (n) = (7 - 1) x (11 - 1)

φ (n) = 6 x 10

φ (n) = 60

Let us now choose the relative prime e of 60 as 7.

Thus, the public key is <e, n> = (7, 77)

Step 4: A plaintext message m is encrypted using public key <e, n>. To find ciphertext from the plain text, the following formula is used to get ciphertext C.

To find ciphertext from the plain text, the following formula is used to get ciphertext C.

C = me mod n

C = 97 mod 77

C = 37

Step 5: The private key is <d, n>. To determine the private key, we use the following formula d such that:

De mod {(p - 1) x (q - 1)} = 1

7d mod 60 = 1, which gives d = 43

The private key is <d, n> = (43, 77)

Step 6: A ciphertext message c is decrypted using the private key <d, n>. To calculate plain text m from the ciphertext c the following formula is used to get plain text m.

m = cd mod n

m = 3743 mod 77

m = 9

In this example, Plain text = 9 and the ciphertext = 37.

II(D). ROBBIN MILLER PRIMALITY TEST:

The Miller-Rabin primality test is a probabilistic algorithm used to determine if a given number is a probable prime or definitely composite. It works based on the properties of Fermat's Little Theorem. The algorithm performs a series of modular exponentiation and probabilistic checks to assess whether a number is likely to be prime. It is widely used in practice due to its efficiency and effectiveness, even though it might have a small probability of error.

Here are the steps:

1. Input: Choose a candidate prime n to be tested for primality. Also, choose a parameter k that determines the accuracy of the test. A higher k value increases the accuracy but also the computational cost.

2. Factorization of n-1: Write n-1 as 2s.d, where s is the largest power of 2 dividing n-1, and d is an odd number.

3. Witness Loop: Repeat the following steps k times

a. Choose a Witness: Select a random integer a such that 2 <= a <= n.

b. Compute ad mod n: Calculate x = ad mod n.

c. Check Conditions: If x = 1 mod n or x = -1 mod n, then continue to the next

Iteration. Otherwise, proceed to the next step.

d. Square-and-Multiply: For r = 1 to s-1, compute x =x^2 mod n. If x = 1 mod n,

then n is composite. If x = -1 mod n, break out of the loop. -

e. Final Check: If x! = -1 mod n, then n is definitely composite.

4. Conclusion: If, after k iterations, n has passed all the tests, it is considered a

probable prime.

## II(E). Advantages of RSA:

* No Key Sharing: RSA encryption depends on using the receiver’s public key, so you don’t have to share any secret key to receive messages from others.
* Proof of Authenticity: Since the key pairs are related to each other, a receiver can’t intercept the message since they won’t have the correct private key to decrypt the information.
* Faster Encryption: The encryption process is faster than that of the DSA algorithm.
* Data Can’t Be Modified: Data will be tamper-proof in transit, since meddling with the data will alter the usage of the keys. And the private key won’t be able to decrypt the information, hence alerting the receiver of manipulation.

II(F). Analysis of RSA:

**Brute-force analysis of RSA algorithm:**

|  |  |
| --- | --- |
| **Bit Size** | **Time taken (in milliseconds)** |
| **0** | **0** |
| **10** | **0.5** |
| **15** | **0.95** |
| **20** | **23.73** |
| **25** | **394.78** |
| **30** | **15985.57** |
| **31** | **29522.25** |
| **32** | **44630.68** |
| **33** | **110705.20** |
| **35** | **1184805.80** |

***Cryptography*** is the science and art of designing and using codes to secure information and communications. The word cryptography comes from the Greek words Krypto’s, meaning hidden, and graphene, meaning writing. Cryptography has a long and fascinating history, dating back to ancient times, when people used simple substitution and transposition ciphers to conceal their messages. Today, cryptography has evolved into a complex and sophisticated field that relies on mathematics, computer science, engineering and physics.

The *main goal of cryptography* is to ensure the confidentiality, integrity, authenticity and availability of data and users. Confidentiality means that only the intended recipients can access the data. Integrity means that the data is not altered or corrupted during transmission or storage. Authenticity means that the data comes from a verified source and has not been forged or impersonated. Availability means that the data is accessible and usable when needed.

To achieve these goals, cryptography uses two main processes: encryption and decryption. Encryption is the process of transforming plain text (the original data) into cipher text (the coded data) using a key (a secret parameter). Decryption is the reverse process of recovering plain text from cipher text using the same or a different key. There are different types of encryption and decryption methods, such as symmetric key, asymmetric key and hash functions.

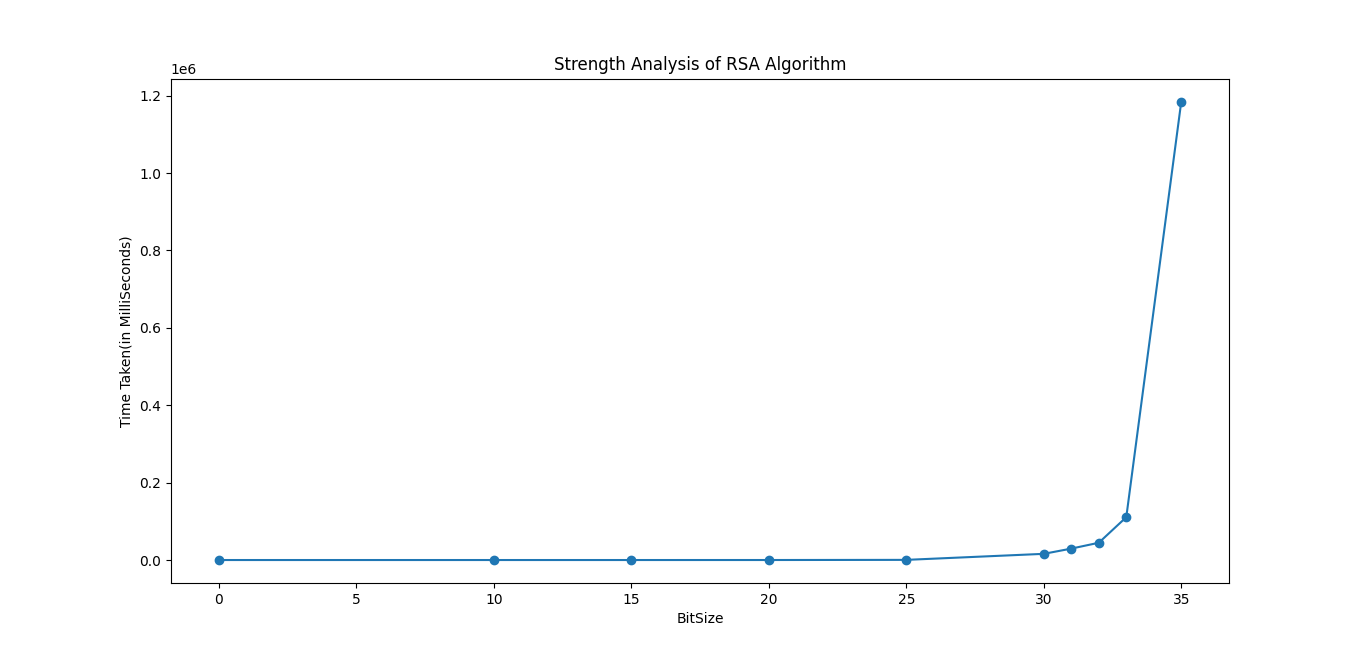
Cryptography has many applications in various domains, such as computer security, digital currencies, web browsing, authentication and cryptocurrencies. Cryptography helps to protect data and users from unauthorized access, tampering, theft and fraud. Cryptography also enables new forms of communication, collaboration and commerce that are based on trust, privacy and security. ****

Figure – 2.3

**(III) ELLIPTIC CURVE CRYPTOGRAPHY (ECC)**

III(A). What is elliptical curve cryptography (ECC)?

Elliptical curve cryptography (ECC) is a public key encryption technique based on elliptic curve theory that can be used to create faster, smaller and more efficient cryptographic keys.

ECC is an alternative to the Rivest-Shamir-Adleman (RSA) cryptographic algorithm and is most often used for digital signatures in cryptocurrencies, such as Bitcoin and Ethereum, as well as one-way encryption of emails, data and software.

An elliptic curve is not an ellipse, or oval shape, but it is represented as a looping line intersecting two axes, which are lines on a graph used to indicate the position of a point. The curve is completely symmetric, or mirrored, along the x-axis of the graph.

Public key cryptography systems, like ECC, use a mathematical process to merge two distinct keys and then use the output to encrypt and decrypt data. One is a public key that is known to anyone, and the other is a private key that is only known by the sender and receiver of the data.

ECC generates keys through the properties of an elliptic curve equation instead of the traditional method of generation as the product of large prime numbers. From a cryptographic perspective, the points along the graph can be formulated using the following equation:

y²=x³ + ax + b

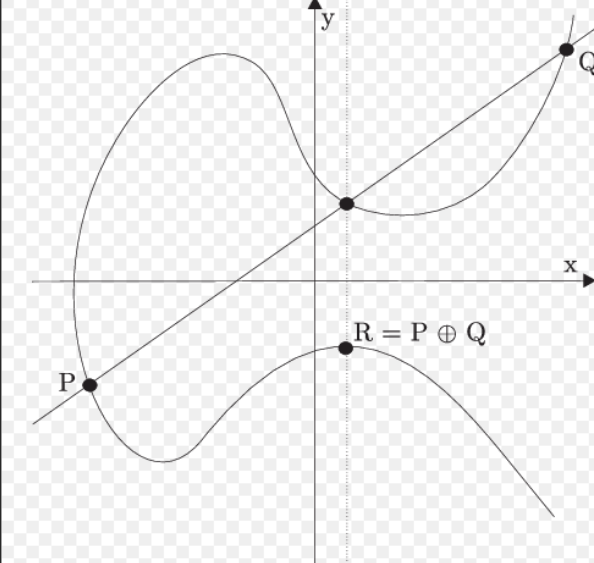


Figure – 3.1

ECC is like most other public key encryption methods, such as the RSA algorithm and Diffie-Hellman. Each of these cryptography mechanisms uses the concept of a one-way, or trapdoor, function. This means that a mathematical equation with a public and private key can be used to easily get from point A to point B. But, without knowing the private key and depending on the key size used, getting from B to A is difficult, if not impossible, to achieve.

ECC is based on the properties of a set of values for which operations can be performed on any two members of the group to produce a third member, which is derived from points where the line intersects the axes as shown with the green line and three blue dots in the below diagram labeled A, B and C. Multiplying a point on the curve by a number produces another point on the curve (C). Taking point C and bringing it to the mirrored point on the opposite side of the x-axis produces point D. From here, a line is drawn back to our original point A, creating an intersection at point E. This process can be completed n number of times within a defined max value. The n is the private key value, which indicates how many times the equation should be run, ending on the final value that is used to encrypt and decrypt data. The maximum defined value of the equation relates to the key size used.

III(B). FUNCTIONS AND ALGORITHM USED IN ECC:

* ***Domain Parameters:***

Elliptic curve domain parameters are the values that define an elliptic curve for cryptography. There are different standards that specify these parameters for various named curves, such as SEC, NIST FIPS PUB 186-4 and Brain pool ECC. However, some cryptographers argue that these standards are not secure enough and propose their own standards, such as Safe Curves.

* ***Point addition:***

Given two points P and Q on an elliptic curve E, draw a line through them and find the third point of intersection with E. Then reflect this point across the x-axis to get the result of P + Q. If P and Q are the same point, this is called point doubling and the line is the tangent to E at P.

* ***Point multiplication:***

Given a point P on an elliptic curve E and an integer k, compute kP by adding P to itself k times. This can be done efficiently by using binary representation of k and applying repeated doubling and addition. For example, if k = 11, then kP = P + 2P + 8P = P + (P + P) + ((P + P) + (P + P)) + (((P + P) + (P + P)) + ((P + P) + (P + P))).

* ***One Way Function in ECC:***

A one-way function in ECC is a function that is easy to compute in one direction, but hard to invert in the other direction. For example, given a point P on an elliptic curve E and an integer k, it is easy to compute kP by point multiplication, but given kP and P, it is hard to find k by solving the discrete logarithm problem. This property makes ECC suitable for public-key cryptography, where the public key can be derived from the private key by a one-way function, but the private key cannot be easily recovered from the public key. ECC offers more security than other public-key schemes like RSA, because it requires smaller key sizes for the same level of security. For example, a 256-bit key in ECC is equivalent to a 3072-bit key in RSA. This means that ECC can save storage space, bandwidth and computational resources.

* ***ECC keys:***

1. *Private key:* ECC cryptography’s private key creation is as simple as safely producing a random integer in a specific range, making it highly quick. Any integer in the field represents a valid ECC private key.
2. *Public keys:* Public keys within ECC are EC points, which are pairs of integer coordinates x, and y that lie on a curve. Because of its unique features, EC points can be compressed to a single coordinate + 1 bit (odd or even). As a result, the compressed public key corresponds to a 256-bit ECC.

* ***Generator Point:***

ECC cryptosystems establish a special pre-defined EC point called generator point G (base point) for elliptic curves over finite fields, which can generate any other position in its subgroup over the elliptic curve by multiplying G from some integer in the range [0…r].

The number r is referred to as the “ordering” of the cyclic subgroup.

Elliptic curve subgroups typically contain numerous generator points, but cryptologists carefully select one of them to generate the entire group (or subgroup), and is excellent for performance optimizations in calculations. This is the “G” generator.

III(C). ELLIPTIC CURVE DIFFIE-HELLMAN ALGORITHM (ECDH)

The ECDH algorithm is a way of securely exchanging keys between two parties over an insecure channel. It is based on the idea of using elliptic curves, which are mathematical curves that have special properties for cryptography. The ECDH algorithm allows two parties, Alice and Bob, to agree on a secret key that only they know, without revealing their private keys to each other or to anyone else. The secret key can then be used for encryption or authentication purposes.

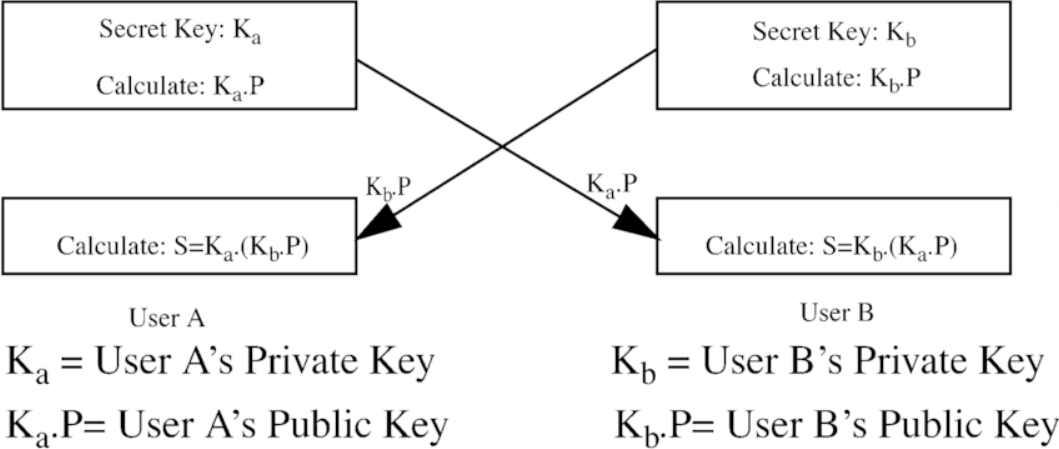


Figure – 3.2

The steps of Implementing ECDH is given below, considering the upper diagram:

* ***Key agreement algorithm*:** This is a protocol that allows two parties, Alice and Bob, to establish a shared secret over an insecure channel. The shared secret can be used as a key or to derive another key for encryption or authentication purposes. The steps for this algorithm are:

1. Alice and Bob agree on an elliptic curve E and a generator point G on E.
2. Alice chooses a random integer a as her private key and computes A = aG as her public key.
3. Bob chooses a random integer b as his private key and computes B = bG as his public key.
4. Alice sends A to Bob and Bob sends B to Alice over the insecure channel.
5. Alice computes S = aB as the shared secret.
6. Bob computes S = bA as the shared secret.

* ***Mathematical explanation:*** Alice and Bob can compute the same shared secret S because of the commutative property of point multiplication on elliptic curves: aB = abG = baG = bA. This means that multiplying the other party’s public key by their own private key gives the same result as multiplying their own public key by the other party’s private key. This is also known as the Diffie-Hellman problem, which is hard to solve without knowing the private keys.

The ECDH algorithm is secure because it is hard to find the private keys a and b from the public keys A and B, or to find the secret key S from A and B, without solving the discrete logarithm problem on elliptic curves, which is believed to be computationally infeasible. The ECDH algorithm also offers more security than other key exchange algorithms like RSA, because it requires smaller key sizes for the same level of security.

III(D). ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM (ECDSA)

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a way of creating and verifying digital signatures using elliptic curves, which are mathematical curves that have special properties for cryptography. Digital signatures are used to prove the authenticity and integrity of messages or data, such as transactions in blockchains or certificates in secure communications.

The ECDSA works as follows:

* ***Step1-Choose curve and generator point:*** Alice and Bob agree on an elliptic curve E and a generator point G on E. These are public parameters that anyone can know.
* ***Step2-Generate keys****:* Alice chooses a random integer k as her private key and computes K = kG as her public key. She publishes her public key to Bob and anyone else who wants to verify her signatures.
* ***Step3- Signing:***

1. Choose a random integer r and compute R = rG.
2. Compute the hash value h = H(m) of the message m.
3. Compute s = (h + rk) / r mod n, where k is the private key and n is the order of G.
4. The signature is the pair (R, s).

* ***Step4- verifying:***

1. Compute the hash value h = H(m) of the message m.
2. Compute u = h / s mod n and v = R / s mod n, where s is the signature component and n are the order of G.
3. Compute U = uG and V = vK, where G is the generator point and K is the public key.
4. Check if U + V = R, where R is the signature component. If yes, the signature is valid. If no, the signature is invalid.

The ECDSA is secure because it is hard to find the private key k from the public key K, or to forge a signature without knowing k, without solving the discrete logarithm problem on elliptic curves, which is believed to be computationally infeasible. The ECDSA also offers more security than other digital signature algorithms like RSA, because it requires smaller key sizes for the same level of security.

**(VI) ECC VS RSA**

IV(A). Why ECC is better than RSA?

Elliptic Curve Cryptography (ECC) is a type of public key cryptography that uses the algebraic structure of elliptic curves to generate keys and perform encryption and decryption operations. RSA is another type of public key cryptography that uses the mathematical problem of factoring large numbers to secure its keys and operations.

One of the main advantages of ECC over RSA is that it can achieve the same level of security with much smaller key sizes. This means that ECC keys are faster to generate, use less storage space, and consume less bandwidth and power when transmitted or processed. For example, a 256-bit ECC key is said to provide security equivalent to a 3072-bit RSA key. This is because the underlying mathematical problem of ECC, which is the discrete logarithm problem on elliptic curves, is harder to solve than the factoring problem of RSA, for a given key size.

The discrete logarithm problem on elliptic curves is also known as the trapdoor function of ECC, because it is easy to compute in one direction but hard to invert in the other direction. For example, given an elliptic curve E and a point G on E, it is easy to compute kG for any integer k, but it is hard to find k given kG. This property makes ECC keys secure, because an attacker cannot easily recover the private key from the public key without solving this problem. The factoring problem of RSA has a similar property, but it is easier to solve with current algorithms and computing power.

IV(B). Different Use cases of Both ECC and RSA:

ECC and RSA are both widely used encryption algorithms, but they have different strengths and weaknesses depending on the use case. Here is a small paragraph about which algorithm is better in which particular cases:

* ECC is better for use cases that require high security, low power consumption, and fast performance. For example, ECC is suitable for mobile devices, IoT devices, and web servers that need to handle numerous SSL/TLS connections. ECC can provide the same level of security as RSA with much smaller key sizes, which means less computation, storage, and bandwidth overhead.
* RSA is better for use cases that require simple implementation, compatibility, and signature verification. For example, RSA is suitable for legacy systems, embedded systems, and digital signatures that need to be verified frequently or by low-power devices. RSA is based on simple mathematical principles and has more widespread support than ECC. RSA also has an advantage over ECC in signature verification speed, which can be important for some applications.

IV(C). Key Strength of ECC vs RSA:

Security depends on the specific algorithm and key length. In the below table, there is a clear comparison of RSA and ECC algorithms that shows how key length increase over a period due to upgrade in computer software and hardware combination. The reason behind choosing ECC for organizations is a shorter key used against lengthy RSA keys.

**(V) Annexure**

V(A). RSA CODE FOR ENCRYPTION AND DECRYPTION AND BRUITEFORRCE ANALYSIS IN JAVA:

import java.util.\*;

import java.math.BigInteger;

import java.security.SecureRandom;

public class Main {

static Scanner sc = new Scanner (System.in);

BigInteger key1;

BigInteger key2;

BigInteger n;

int bitSize;

SecureRandom random = new SecureRandom();

public BigInteger generatePrime() {

bitSize = 35;// ?34

while (true) {

BigInteger randomBigInteger = new BigInteger(bitSize, random);

if (randomBigInteger.isProbablePrime(15)) {

return randomBigInteger;

}

}

}

public void setKeys() {

BigInteger prime1 = generatePrime();

BigInteger prime2 = generatePrime();

while (prime1.equals(prime2)) {

prime2 = generatePrime();

}

System.out.println("Prime 1: " + prime1.toString());

System.out.println("Prime 2: " + prime2.toString());

n = prime1.multiply(prime2);

// System.out.println("N: " + n.toString());

BigInteger phi = (prime1.subtract(new BigInteger("1"))).multiply(prime2.subtract(new BigInteger("1")));

BigInteger e = new BigInteger("2");

while (true) {

if ((e.gcd(phi)).compareTo(new BigInteger("1")) == 0)

break;

e = e.add(new BigInteger("1"));

}

// System.out.println("e : " + e.toString());

key1 = e;

BigInteger d = e.modInverse(phi);

// System.out.println("d : " + d.toString());

key2 = d;

}

public BigInteger encrypt(BigInteger message) {

return message.modPow(key1, n);

}

public BigInteger decrypt(BigInteger encryptedText) {

return encryptedText.modPow(key2, n);

}

public List<BigInteger> encoder(String message) {

List<BigInteger> msg = new ArrayList<>();

for (char letter : message.toCharArray()) {

BigInteger charValue = BigInteger.valueOf((long) letter);

BigInteger encryptedChar = encrypt(charValue);

msg.add(encryptedChar);

}

return msg;

}

public String decoder(List<BigInteger> encoded) {

StringBuilder s = new StringBuilder();

for (BigInteger encryptedChar : encoded) {

BigInteger decryptedChar = decrypt(encryptedChar);

s.append((char) decryptedChar.longValue());

}

return s.toString();

}

// !!!BREAK

public void StrengthTest() {

long ans = 0;

long ans1 = 0;

int i = 0;

while (i != 10) {

System.out.println((i + 1) + " : ");

setKeys();

String s = "Checking The Strength of Algorithm";

long startTime1 = System.currentTimeMillis();

List<BigInteger> v = encoder(s);

long endTime1 = System.currentTimeMillis();

long startTime = System.currentTimeMillis();

breakEncryption(n, key1, v, s);

long endTime = System.currentTimeMillis();

i++;

ans += (endTime - startTime);

ans1 += (endTime1 - startTime1);

}

System.out.println(bitSize + " --> " + (ans));

System.out.println(bitSize + " --> " + (ans1));

}

BigInteger decrypt(BigInteger encrpyted\_text, BigInteger key2, BigInteger n) {

BigInteger d = key2;

BigInteger decrypted = encrpyted\_text.modPow(d, n);

return decrypted;

}

public String decoder(List<BigInteger> encoded, BigInteger d, BigInteger n) {

StringBuilder s = new StringBuilder();

for (BigInteger encryptedChar : encoded) {

BigInteger decryptedChar = decrypt(encryptedChar, d, n);

s.append((char) decryptedChar.longValue());

}

return s.toString();

}

void breakEncryption(BigInteger n, BigInteger e, List<BigInteger> cipher, String txt) {

// ArrayList<BigInteger> pr = generatePrimes(n);

// long p = pr.get(0);

// long q = pr.get(1);

// for(BigInteger i=new BigInteger("2");)

// System.out.println("N: " + n.toString());

BigInteger p = new BigInteger("2");

while (p.compareTo(n) <= 0) {

if (n.remainder(p).equals(BigInteger.ZERO)) {

break;

// System.out.println("Prime divisor found: " + divisor);

// number = number.divide(divisor);

} else {

p = p.add(BigInteger.ONE);

}

}

// System.out.println("P : " + p.toString());

BigInteger q = n.divide(p);

// System.out.println("Q : " + q.toString());

// q = n.divide(p);

// System.out.println(p + " " + q);

// if (p == n || q == n) {

// System.out.println("Error");

// return;

// }

BigInteger phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

BigInteger k = new BigInteger("1");

BigInteger d;

String s;

BigInteger nm;

while (true) {

nm = BigInteger.ONE.add(k.multiply(phi));

d = nm.divide(e);

// System.out.println("d : " + d);

s = decoder(cipher, d, n);

if (s.equals(txt)) {

// System.out.println(d + " " + k);

// System.out.println("Text : " + s);

break;

}

k = k.add(BigInteger.ONE);

}

}

public static void main(String[] args) {

Main rsa = new Main();

rsa.StrengthTest();

// rsa.setKeys();

// /\*

// \* String s = "Hi There";

// \* List<BigInteger> v = rsa.encoder(s);

// \* System.out.println("Encrypted Message : ");

// \* v.forEach((elem) -> System.out.print(elem + " "));

// \* System.out.println("\nDecrypted Message : ");

// \* String sh = rsa.decoder(v);

// \* System.out.println(sh);

// \*/

// // !Custom Check

// System.out.println("Enter Text to be Encrypted : ");

// String s = sc.nextLine();

// List<BigInteger> v = rsa.encoder(s);

// System.out.println("\nEncrypted Message : ");

// v.forEach((elem) -> System.out.print(elem + " "));

// System.out.println("\n\nDecrypted Message : ");

// String sh = rsa.decoder(v);

// rsa.StrengthTest(v);

// System.out.println(sh);

}

}

V(B). ECC CODE FOR ENCRYPTION AND DECRYPTION IN PYTHON:

# Importing required libraries used

# to perform arithmetic operations

# on elliptic curves

# from tinyec import registry

# -\*- coding: utf-8 -\*-

import random

import warnings

# Python3 compatibility

try:

LONG\_TYPE = long

except NameError:

LONG\_TYPE = int

def randbelow(exclusive\_upper\_bound):

"""Return a random int in the range [0, n)."""

if exclusive\_upper\_bound <= 0:

raise ValueError("Upper bound must be positive.")

return random.SystemRandom().\_randbelow(exclusive\_upper\_bound)

def egcd(a, b):

if a == 0:

return b, 0, 1

else:

g, y, x = egcd(b % a, a)

return g, x - (b // a) \* y, y

def mod\_inv(a, p):

if a < 0:

return p - mod\_inv(-a, p)

g, x, y = egcd(a, p)

if g != 1:

raise ArithmeticError("Modular inverse does not exist")

else:

return x % p

class Curve(object):

def \_\_init\_\_(self, a, b, field, name="undefined"):

self.name = name

self.a = a

self.b = b

self.field = field

self.g = Point(self, self.field.g[0], self.field.g[1])

def is\_singular(self):

return (4 \* self.a\*\*3 + 27 \* self.b\*\*2) % self.field.p == 0

def on\_curve(self, x, y):

return (y\*\*2 - x\*\*3 - self.a \* x - self.b) % self.field.p == 0

def \_\_eq\_\_(self, other):

if not isinstance(other, Curve):

return False

return self.a == other.a and self.b == other.b and self.field == other.field

def \_\_ne\_\_(self, other):

return not self.\_\_eq\_\_(other)

def \_\_str\_\_(self):

return "\"%s\" => y^2 = x^3 + %dx + %d (mod %d)" % (self.name, self.a,

self.b, self.field.p)

class SubGroup(object):

def \_\_init\_\_(self, p, g, n, h):

self.p = p

self.g = g

self.n = n

self.h = h

def \_\_eq\_\_(self, other):

if not isinstance(other, SubGroup):

return False

return self.p == other.p and self.g == other.g and self.n == other.n and self.h == other.h

def \_\_ne\_\_(self, other):

return not self.\_\_eq\_\_(other)

def \_\_str\_\_(self):

return "Subgroup => generator %s, order: %d, cofactor: %d on Field => prime %d" % (

self.g, self.n, self.h, self.p)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

class Inf(object):

def \_\_init\_\_(self, curve, x=None, y=None):

self.x = x

self.y = y

self.curve = curve

def \_\_eq\_\_(self, other):

if not isinstance(other, Inf):

return False

return self.curve == other.curve

def \_\_ne\_\_(self, other):

return not self.\_\_eq\_\_(other)

def \_\_add\_\_(self, other):

if isinstance(other, Inf):

return Inf()

if isinstance(other, Point):

return other

raise TypeError("Unsupported operand type(s) for +: '%s' and '%s'" %

(other.\_\_class\_\_.\_\_name\_\_, self.\_\_class\_\_.\_\_name\_\_))

def \_\_sub\_\_(self, other):

if isinstance(other, Inf):

return Inf()

if isinstance(other, Point):

return other

raise TypeError("Unsupported operand type(s) for +: '%s' and '%s'" %

(other.\_\_class\_\_.\_\_name\_\_, self.\_\_class\_\_.\_\_name\_\_))

def \_\_str\_\_(self):

return "%s on %s" % (self.\_\_class\_\_.\_\_name\_\_, self.curve)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

class Point(object):

def \_\_init\_\_(self, curve, x, y):

self.curve = curve

self.x = x

self.y = y

self.p = self.curve.field.p

self.on\_curve = True

if not self.curve.on\_curve(self.x, self.y):

warnings.warn("Point (%d, %d) is not on curve \"%s\"" %

(self.x, self.y, self.curve))

self.on\_curve = False

def \_\_m(self, p, q):

if p.x == q.x:

return (3 \* p.x\*\*2 + self.curve.a) \* mod\_inv(2 \* p.y, self.p)

else:

return (p.y - q.y) \* mod\_inv(p.x - q.x, self.p)

def \_\_eq\_\_(self, other):

if not isinstance(other, Point):

return False

return self.x == other.x and self.y == other.y and self.curve == other.curve

def \_\_ne\_\_(self, other):

return not self.\_\_eq\_\_(other)

def \_\_add\_\_(self, other):

if isinstance(other, Inf):

return self

if isinstance(other, Point):

if self.x == other.x and self.y != other.y:

return Inf(self.curve)

elif self.curve == other.curve:

m = self.\_\_m(self, other)

x\_r = (m\*\*2 - self.x - other.x) % self.p

y\_r = -(self.y + m \* (x\_r - self.x)) % self.p

return Point(self.curve, x\_r, y\_r)

else:

raise ValueError("Cannot add points belonging to different curves")

else:

raise TypeError("Unsupported operand type(s) for +: '%s' and '%s'" %

(other.\_\_class\_\_.\_\_name\_\_, self.\_\_class\_\_.\_\_name\_\_))

def \_\_sub\_\_(self, other):

if isinstance(other, Inf):

return self.\_\_add\_\_(other)

if isinstance(other, Point):

return self.\_\_add\_\_(Point(self.curve, other.x, -other.y % self.p))

else:

raise TypeError("Unsupported operand type(s) for -: '%s' and '%s'" %

(other.\_\_class\_\_.\_\_name\_\_, self.\_\_class\_\_.\_\_name\_\_))

def \_\_mul\_\_(self, other):

if isinstance(other, Inf):

return Inf(self.curve)

if isinstance(other, int) or isinstance(other, LONG\_TYPE):

if other % self.curve.field.n == 0:

return Inf(self.curve)

if other < 0:

addend = Point(self.curve, self.x, -self.y % self.p)

else:

addend = self

result = Inf(self.curve)

# Iterate over all bits starting by the LSB

for bit in reversed([int(i) for i in bin(abs(other))[2:]]):

if bit == 1:

result += addend

addend += addend

return result

else:

raise TypeError("Unsupported operand type(s) for \*: '%s' and '%s'" %

(other.\_\_class\_\_.\_\_name\_\_, self.\_\_class\_\_.\_\_name\_\_))

def \_\_rmul\_\_(self, other):

return self.\_\_mul\_\_(other)

def \_\_str\_\_(self):

return "(%d, %d) %s %s" % (self.x, self.y,

"on" if self.on\_curve else "off", self.curve)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def make\_keypair(curve):

priv = random.randint(1, curve.field.n)

pub = priv \* curve.g

return Keypair(curve, priv, pub)

class Keypair(object):

def \_\_init\_\_(self, curve, priv=None, pub=None):

if priv is None and pub is None:

raise ValueError("Private and/or public key must be provided")

self.curve = curve

self.can\_sign = True

self.can\_encrypt = True

if priv is None:

self.can\_sign = False

self.priv = priv

self.pub = pub

if pub is None:

self.pub = self.priv \* self.curve.g

class ECDH(object):

def \_\_init\_\_(self, keypair):

self.keypair = keypair

def get\_secret(self, keypair):

# Don;t check if both keypairs are on the same curve. Should raise a warning only

if self.keypair.can\_sign and keypair.can\_encrypt:

secret = self.keypair.priv \* keypair.pub

elif self.keypair.can\_encrypt and keypair.can\_sign:

secret = self.keypair.pub \* keypair.priv

else:

raise ValueError("Missing crypto material to generate DH secret")

return secret

EC\_CURVE\_REGISTRY = {

"brainpoolP160r1": {

"p":

0xE95E4A5F737059DC60DFC7AD95B3D8139515620F,

"a":

0x340E7BE2A280EB74E2BE61BADA745D97E8F7C300,

"b":

0x1E589A8595423412134FAA2DBDEC95C8D8675E58,

"g": (0xBED5AF16EA3F6A4F62938C4631EB5AF7BDBCDBC3,

0x1667CB477A1A8EC338F94741669C976316DA6321),

"n":

0xE95E4A5F737059DC60DF5991D45029409E60FC09,

"h":

0x1

},

"brainpoolP192r1": {

"p":

0xC302F41D932A36CDA7A3463093D18DB78FCE476DE1A86297,

"a":

0x6A91174076B1E0E19C39C031FE8685C1CAE040E5C69A28EF,

"b":

0x469A28EF7C28CCA3DC721D044F4496BCCA7EF4146FBF25C9,

"g": (0xC0A0647EAAB6A48753B033C56CB0F0900A2F5C4853375FD6,

0x14B690866ABD5BB88B5F4828C1490002E6773FA2FA299B8F),

"n":

0xC302F41D932A36CDA7A3462F9E9E916B5BE8F1029AC4ACC1,

"h":

0x1

},

"brainpoolP224r1": {

"p":

0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF,

"a":

0x68A5E62CA9CE6C1C299803A6C1530B514E182AD8B0042A59CAD29F43,

"b":

0x2580F63CCFE44138870713B1A92369E33E2135D266DBB372386C400B,

"g": (0x0D9029AD2C7E5CF4340823B2A87DC68C9E4CE3174C1E6EFDEE12C07D,

0x58AA56F772C0726F24C6B89E4ECDAC24354B9E99CAA3F6D3761402CD),

"n":

0xD7C134AA264366862A18302575D0FB98D116BC4B6DDEBCA3A5A7939F,

"h":

0x1

},

"brainpoolP256r1": {

"p":

0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377,

"a":

0x7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B44F330B5D9,

"b":

0x26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6,

"g":

(0x8BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1E3BD23C23A4453BD9ACE3262,

0x547EF835C3DAC4FD97F8461A14611DC9C27745132DED8E545C1D54C72F046997),

"n":

0xA9FB57DBA1EEA9BC3E660A909D838D718C397AA3B561A6F7901E0E82974856A7,

"h":

0x1

},

"brainpoolP320r1": {

"p":

0xD35E472036BC4FB7E13C785ED201E065F98FCFA6F6F40DEF4F92B9EC7893EC28FCD412B1F1B32E27,

"a":

0x3EE30B568FBAB0F883CCEBD46D3F3BB8A2A73513F5EB79DA66190EB085FFA9F492F375A97D860EB4,

"b":

0x520883949DFDBC42D3AD198640688A6FE13F41349554B49ACC31DCCD884539816F5EB4AC8FB1F1A6,

"g":

(0x43BD7E9AFB53D8B85289BCC48EE5BFE6F20137D10A087EB6E7871E2A10A599C710AF8D0D39E20611,

0x14FDD05545EC1CC8AB4093247F77275E0743FFED117182EAA9C77877AAAC6AC7D35245D1692E8EE1

),

"n":

0xD35E472036BC4FB7E13C785ED201E065F98FCFA5B68F12A32D482EC7EE8658E98691555B44C59311,

"h":

0x1

},

"brainpoolP384r1": {

"p":

0x8CB91E82A3386D280F5D6F7E50E641DF152F7109ED5456B412B1DA197FB71123ACD3A729901D1A71874700133107EC53,

"a":

0x7BC382C63D8C150C3C72080ACE05AFA0C2BEA28E4FB22787139165EFBA91F90F8AA5814A503AD4EB04A8C7DD22CE2826,

"b":

0x04A8C7DD22CE28268B39B55416F0447C2FB77DE107DCD2A62E880EA53EEB62D57CB4390295DBC9943AB78696FA504C11,

"g":

(0x1D1C64F068CF45FFA2A63A81B7C13F6B8847A3E77EF14FE3DB7FCAFE0CBD10E8E826E03436D646AAEF87B2E247D4AF1E,

0x8ABE1D7520F9C2A45CB1EB8E95CFD55262B70B29FEEC5864E19C054FF99129280E4646217791811142820341263C5315

),

"n":

0x8CB91E82A3386D280F5D6F7E50E641DF152F7109ED5456B31F166E6CAC0425A7CF3AB6AF6B7FC3103B883202E9046565,

"h":

0x1

},

"brainpoolP512r1": {

"p":

0xAADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639CCA703308717D4D9B009BC66842AECDA12AE6A380E62881FF2F2D82C68528AA6056583A48F3,

"a":

0x7830A3318B603B89E2327145AC234CC594CBDD8D3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9AC8B57F1117A72BF2C7B9E7C1AC4D77FC94CA,

"b":

0x3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9AC8B57F1117A72BF2C7B9E7C1AC4D77FC94CADC083E67984050B75EBAE5DD2809BD638016F723,

"g":

(0x81AEE4BDD82ED9645A21322E9C4C6A9385ED9F70B5D916C1B43B62EEF4D0098EFF3B1F78E2D0D48D50D1687B93B97D5F7C6D5047406A5E688B352209BCB9F822,

0x7DDE385D566332ECC0EABFA9CF7822FDF209F70024A57B1AA000C55B881F8111B2DCDE494A5F485E5BCA4BD88A2763AED1CA2B2FA8F0540678CD1E0F3AD80892

),

"n":

0xAADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639CCA70330870553E5C414CA92619418661197FAC10471DB1D381085DDADDB58796829CA90069,

"h":

0x1

},

"secp192r1": {

"p":

0xfffffffffffffffffffffffffffffffeffffffffffffffff,

"a":

0xfffffffffffffffffffffffffffffffefffffffffffffffc,

"b":

0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1,

"g": (0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012,

0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811),

"n":

0xffffffffffffffffffffffff99def836146bc9b1b4d22831,

"h":

0x1

},

"secp224r1": {

"p":

0xffffffffffffffffffffffffffffffff000000000000000000000001,

"a":

0xfffffffffffffffffffffffffffffffefffffffffffffffffffffffe,

"b":

0xb4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4,

"g": (0xb70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21,

0xbd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34),

"n":

0xffffffffffffffffffffffffffff16a2e0b8f03e13dd29455c5c2a3d,

"h":

0x1

},

"secp256r1": {

"p":

0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff,

"a":

0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc,

"b":

0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b,

"g":

(0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296,

0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5),

"n":

0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551,

"h":

0x1

},

"secp384r1": {

"p":

0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffffffff0000000000000000ffffffff,

"a":

0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffffffff0000000000000000fffffffc,

"b":

0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef,

"g":

(0xaa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a385502f25dbf55296c3a545e3872760ab7,

0x3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c00a60b1ce1d7e819d7a431d7c90ea0e5f

),

"n":

0xffffffffffffffffffffffffffffffffffffffffffffffffc7634d81f4372ddf581a0db248b0a77aecec196accc52973,

"h":

0x1

},

"secp521r1": {

"p":

0x000001ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff,

"a":

0x000001fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc,

"b":

0x00000051953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451fd46b503f00,

"g":

(0x000000c6858e06b70404e9cd9e3ecb662395b4429c648139053fb521f828af606b4d3dbaa14b5e77efe75928fe1dc127a2ffa8de3348b3c1856a429bf97e7e31c2e5bd66,

0x0000011839296a789a3bc0045c8a5fb42c7d1bd998f54449579b446817afbd17273e662c97ee72995ef42640c550b9013fad0761353c7086a272c24088be94769fd16650

),

"n":

0x000001fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffa51868783bf2f966b7fcc0148f709a5d03bb5c9b8899c47aebb6fb71e91386409,

"h":

0x1

}

}

def get\_curve(name):

curve\_params = {}

for k, v in EC\_CURVE\_REGISTRY.items():

if name.lower() == k.lower():

curve\_params = v

if curve\_params == {}:

raise ValueError("Unknown elliptic curve name")

try:

sub\_group = SubGroup(curve\_params["p"], curve\_params["g"],

curve\_params["n"], curve\_params["h"])

curve = Curve(curve\_params["a"], curve\_params["b"], sub\_group, name)

except KeyError:

raise RuntimeError("Missing parameters for curve %s" % name)

return curve

# Function to calculate compress point

# of elliptic curves

def compress(publicKey):

return hex(publicKey.x) + hex(publicKey.y % 2)[2:]

# The elliptic curve which is used for the ECDH calculations

curve = get\_curve('brainpoolP256r1')

# Generation of secret key and public key

Ka = randbelow(curve.field.n)

X = Ka \* curve.g

print("X:", compress(X))

Kb = randbelow(curve.field.n)

Y = Kb \* curve.g

print("Y:", compress(Y))

print("\nCurrently exchange the publickey (e.g. through Internet)\n")

# (A\_SharedKey): represents user A

# (B\_SharedKey): represents user B

A\_SharedKey = Ka \* Y

print("(A) shared key :", compress(A\_SharedKey))

B\_SharedKey = Kb \* X

print("(B) shared key :", compress(B\_SharedKey))

print("Equal shared keys:", A\_SharedKey == B\_SharedKey)