CYBERSECURITY & ETHICAL HACKING
CRYPTOGRAPHY ANALYSIS AND IMPLEMENT
ASSIGNMENT - III

KHUSHI JAIN
20BCE2664

I. ANALYSIS

A. SYMMETRIC KEY ALGORITHMS

i. Key Strengths

a) The cryptographic strength of a symmetric key algorithm pertains to its ability to withstand attacks and ensure the confidentiality of the encrypted data. Several factors contribute to the cryptographic strength of a symmetric key algorithm:

b) Key Size: The size of the key used in the algorithm plays a crucial role in its strength. In general, a larger key size enhances the cryptographic strength, as it increases the number of possible key combinations. This makes exhaustive search attacks, such as brute-force attacks, more computationally infeasible.

c) Block Size: The block size of a symmetric key algorithm refers to the length of the data blocks processed by the algorithm during encryption and decryption. A larger block size can enhance security by introducing a higher degree of confusion and diffusion. It helps prevent attacks that exploit patterns in the plaintext or ciphertext.

d) Number of Rounds: The number of rounds indicates the iteration of cryptographic transformations applied within the algorithm. Increasing the number of rounds can enhance security by making the algorithm more resistant to various attacks. Each round introduces additional complexity and makes it harder for an attacker to analyze or exploit the algorithm's structure.

e) Resistance to Attacks: The ability of a symmetric key algorithm to withstand different types of attacks, such as brute-force attacks, differential cryptanalysis, or known-plaintext attacks, significantly influences its cryptographic strength. A strong algorithm should exhibit resilience against known attack techniques, ensuring that it remains secure even when subjected to exhaustive analysis.

ii. Advantages of the Algorithm

a) Efficiency: Symmetric key algorithms are generally faster and more efficient than their asymmetric key counterparts. The encryption and decryption processes involve simple mathematical operations that can be performed quickly, making them suitable for applications that require high-speed data processing.

b) Simplicity: Symmetric key algorithms are relatively straightforward to implement and understand. They involve a single key for both encryption and decryption, simplifying the key management process compared to asymmetric key algorithms.

c) Secure for Bulk Data Encryption: Symmetric key algorithms are well-suited for bulk data encryption, where large amounts of data need to be encrypted or decrypted quickly. Their

efficiency and simplicity make them ideal for securing data storage, transmission, or real-time communication systems.

d) Resource Efficiency: Symmetric key algorithms require fewer computational resources, such as processing power and memory, compared to asymmetric key algorithms. This makes them more suitable for resource-constrained environments, such as embedded systems or mobile devices.

iii. Known Vulnerabilities or weaknesses

a) Key Distribution: The primary challenge with symmetric key algorithms is securely distributing the encryption keys between the communicating parties. Since the same key is used for encryption and decryption, any compromise or interception of the key can lead to a complete breach of security.

b) Key Management: Symmetric key algorithms require effective key management practices to ensure the security of the encryption keys. Key generation, storage, distribution, and revocation procedures must be carefully designed and implemented to prevent unauthorized access to the keys.

c) Lack of Forward Secrecy: In symmetric key algorithms, if the encryption key is compromised, all previously encrypted data can be decrypted. There is no forward secrecy, meaning that a compromise of the key compromises all past and future communications.

d) Scalability: As the number of communicating parties' increases, the key distribution and management complexities grow exponentially in symmetric key algorithms. Sharing a unique key securely with each participant becomes increasingly challenging, which is why symmetric key algorithms are not commonly used for secure communication among a large number of entities.

iv. Real – World Examples of where the Algorithm is commonly used

a) Data Encryption Standard (DES): DES is a widely used symmetric key algorithm. It was the dominant encryption standard for several decades but is now considered weak due to its small key size (56 bits) and vulnerability to brute-force attacks. DES has been largely replaced by stronger symmetric key algorithms like AES (Advanced Encryption Standard).

b) RC4: RC4 is a stream cipher symmetric key algorithm. It gained popularity due to its simplicity and fast execution. However, vulnerabilities have been discovered in RC4, leading to its deprecation and discontinuation in many cryptographic applications.

c) Triple DES (3DES): 3DES is a symmetric key algorithm that applies the DES algorithm three times to each data block. It provides a higher level of security compared to DES, but its efficiency is significantly lower. As a result, it has been largely replaced by AES for secure communications.

d) Advanced Encryption Standard (AES): AES is currently the most widely used symmetric key algorithm. It is highly secure and resistant to various known attacks. AES supports key sizes of 128, 192, and 256 bits, making it suitable for a wide range of applications, including securing sensitive data, communication, and financial transactions.

B. ASYMMETRIC KEY ALGORITHMS

i. Key Strengths

a) Key Exchange and Secure Communication: Asymmetric key algorithms enable secure key exchange without requiring a pre-shared secret key. This property allows secure communication between parties who have not previously shared a secret key. Examples of asymmetric key algorithms used for key exchange include Diffie-Hellman and Elliptic Curve Diffie-Hellman (ECDH).

b) Digital Signatures and Authentication: Asymmetric key algorithms provide robust mechanisms for authentication and digital signatures. Digital signatures verify the integrity and authenticity of messages or documents, ensuring they haven't been tampered with or forged. Asymmetric key algorithms like RSA and Digital Signature Algorithm (DSA) are commonly used for digital signatures.

c) Secure Encryption and Confidentiality: Asymmetric key algorithms can be used for secure encryption, ensuring the confidentiality of data. The recipient's private key is used for decryption, which provides a higher level of security compared to symmetric key algorithms where the same key is used for both encryption and decryption. Examples of asymmetric key encryption algorithms include RSA and Elliptic Curve Cryptography (ECC).

d) Forward Secrecy: Asymmetric key algorithms provide forward secrecy, meaning that even if the private key of a user is compromised, past communications remain secure. Each session can use a unique session key derived from the public and private keys of the participants, ensuring that compromising the current key does not compromise past communications.

ii. Advantages of the Algorithm

a) Key Distribution: Asymmetric key algorithms eliminate the need for secure key distribution. Each participant in a communication pair has a unique key pair consisting of a public key and a private key. The public keys can be freely shared, allowing anyone to encrypt messages, while the corresponding private keys remain secret and are used for decryption. This enables secure communication without the need for prior key exchange.

b) Authentication and Digital Signatures: Asymmetric key algorithms facilitate authentication and digital signatures. The private key associated with a specific public key can be used to generate a digital signature for a message. The signature can be verified using the corresponding public key, ensuring the authenticity and integrity of the message. This capability is valuable for verifying the identity of communicating parties and ensuring the integrity of transmitted data.

c) Forward Secrecy: With asymmetric key algorithms, even if the private key of a user is compromised, it does not compromise the security of previously encrypted messages. Each communication session can have a unique session key generated by combining the public and private keys of the participants. This property provides forward secrecy, meaning that past communications remain secure even if current keys are compromised.

d) Secure Key Exchange: Asymmetric key algorithms can be used for secure key exchange in symmetric key encryption systems. For example, the Diffie-Hellman key exchange protocol allows two parties to generate a shared secret key over an insecure channel without transmitting the key itself. This shared key can then be used for secure communication using symmetric key algorithms.

iii. Known Vulnerabilities or weaknesses

a) Computational Complexity: Asymmetric key algorithms are computationally more intensive compared to symmetric key algorithms. The mathematical operations involved in encryption and decryption using asymmetric keys are more complex, requiring higher computational resources. This can impact performance in resource-constrained environments.

b) Key Length and Key Management: The security of asymmetric key algorithms relies on the length of the keys used. Longer key lengths increase the computational complexity of attacks, making them more secure. However, longer key lengths also require more storage and processing resources. Additionally, key management becomes more challenging as the number of key pairs increases, requiring robust infrastructure for key generation, storage, and distribution.

c) Reliance on Trust and Public Key Infrastructure (PKI): Asymmetric key algorithms rely on the trustworthiness of public keys. Ensuring the authenticity and integrity of public keys is crucial to prevent man-in-the-middle attacks or impersonation. Public Key Infrastructure (PKI) provides mechanisms for managing and verifying public keys, but it requires a trusted infrastructure and proper implementation to be effective.

d) Limited Efficiency for Bulk Data Encryption: Asymmetric key algorithms are not well-suited for bulk data encryption due to their computational complexity. They are typically used for securing key exchange and authentication processes, while actual data encryption is often performed using symmetric key algorithms to achieve higher efficiency.

iv. Real – World Examples of where the Algorithm is commonly used

a) RSA: RSA (Rivest-Shamir-Adleman) is a widely used asymmetric key algorithm for encryption, decryption, and digital signatures. It is based on the mathematical difficulty of factoring large prime numbers. RSA is widely adopted for secure communication, including secure email, SSL/TLS protocols for secure web browsing, and secure file transfer.

b) Elliptic Curve Cryptography (ECC): ECC is an asymmetric key algorithm that uses the mathematics of elliptic curves to provide strong security with relatively shorter key lengths compared to other asymmetric algorithms. ECC is gaining popularity due to its efficiency and suitability for resource-constrained environments like mobile devices.

c) Diffie-Hellman Key Exchange: Diffie-Hellman is a key exchange protocol based on asymmetric key algorithms. It allows two parties to establish a shared secret.

C. HASH FUNCTIONS

i. Key Strengths

a) Data Integrity Verification: Hash functions provide a means to verify the integrity of data. By generating a hash value for a piece of data, any subsequent modification or tampering with the data will result in a different hash value. Comparing the computed hash value with the original hash value allows the recipient to ensure data integrity.

b) Efficient and Fast: Hash functions are designed to be computationally efficient, allowing them to process data quickly and generate hash values in a timely manner. This efficiency makes them suitable for various applications, including checksums, data deduplication, and data indexing.

c) Fixed Output Size: Hash functions produce fixed-size hash values regardless of the size of the input data. This property makes them convenient for storage, comparison, and indexing purposes. The fixed output size also ensures that hash values can be securely transmitted or stored without revealing the original data.

d) Uniqueness and Identifiers: Hash functions are used to generate unique identifiers or keys for data. They are commonly used in data structures like hash tables or hash-based indexing to efficiently retrieve data. Additionally, hash functions play a crucial role in data fingerprinting, digital signatures, and generating identifiers for entities in various cryptographic systems.

ii. Advantages of the Algorithm

a) Data Integrity: Hash functions are commonly used to ensure data integrity. A hash function takes an input (message or data) and produces a fixed-size hash value or digest. Even a small change in the input data results in a significantly different hash value. By comparing the computed hash value with the original hash value, one can verify if the data has been altered or tampered with.

b) Data Verification: Hash functions provide a quick and efficient way to verify the integrity of data during transmission or storage. By generating a hash value for the data and comparing it with a known hash value, the recipient can determine if the data has been modified during transit or storage. This is particularly useful for verifying the authenticity and integrity of downloaded files or verifying the integrity of sensitive data.

c) Password Storage: Hash functions are commonly used for storing passwords securely. Instead of storing the actual passwords, a hash value of the password is stored in a database. When a user enters their password, it is hashed and compared to the stored hash value. This ensures that even if the database is compromised, the actual passwords remain protected.

d) Uniqueness and Identifiers: Hash functions are used to generate unique identifiers or keys for various purposes. For example, hash functions are utilized in hash tables for efficient data retrieval, in digital certificates for generating unique identifiers for entities, and in data indexing and searching algorithms.

iii. Known Vulnerabilities or weaknesses

a) Collision Vulnerability: Hash functions are susceptible to collisions, which occur when two different inputs produce the same hash value. In theory, hash functions should have an extremely low probability of collision. However, as computing power increases, the chances of finding collisions for weaker hash functions also increase. Cryptographically secure hash functions are designed to have a negligible collision probability.

b) Preimage Attacks: Preimage attacks aim to find an input that produces a specific hash value. A hash function is considered secure if it is computationally infeasible to find an input that matches a given hash value. Weaker hash functions may be vulnerable to preimage attacks, where an attacker can find a specific input that results in a desired hash value.

c) Length Extension Attacks: Length extension attacks exploit the vulnerability of some hash functions that allow an attacker to extend the hash value of a given message without knowing the original message. This can lead to potential security breaches, especially if hash functions are used in cryptographic protocols without proper precautions.

d) Dependency on Algorithm Security: The security of hash functions relies on the strength of the underlying algorithm. If a hash function's algorithm is found to have weaknesses or vulnerabilities, it can compromise the security of applications relying on that hash function. Therefore, it is important to use well-studied and widely accepted hash functions that have undergone thorough cryptographic analysis.

iv. Real – World Examples of where the Algorithm is commonly used

a) SHA-2 (Secure Hash Algorithm 2): SHA-2 is a family of cryptographic hash functions that includes SHA-224, SHA-256, SHA-384, and SHA-512. These hash functions are widely used in various applications, including digital signatures, certificates, secure communication protocols, and password storage.

b) MD5 (Message Digest Algorithm 5): MD5 is a widely used hash function for non-cryptographic purposes. However, it is considered cryptographically broken due to its vulnerability to collision attacks. As a result, it is no longer recommended for cryptographic applications but is still used for non-security-sensitive purposes such as checksums and fingerprinting.

c) SHA-3 (Secure Hash Algorithm 3): SHA-3 is the latest member of the Secure Hash Algorithm family. It provides hash functions with improved security and resistance against cryptographic attacks.

II. IMPLEMENTATION STEPS

1. Import the necessary libraries: **cv2**, **numpy**, **glob**, and **matplotlib.pyplot**.
2. Read the images using **cv2.imread()** and store them in a list.
3. Iterate over the images and display each image using **matplotlib.pyplot.imshow()** and **matplotlib.pyplot.show()**.

RSA Algorithm:

1. Generate two large prime numbers, **P** and **Q**, randomly using the **genPrimeNum()** function.
2. Calculate **N** as the product of **P** and **Q**, and **eulerTotient** as **(P - 1) * (Q - 1)**.
3. Find a public exponent **E** such that **GCD(E, eulerTotient) = 1** using the **findPubExp()** function.
4. Use the Extended Euclidean Algorithm (**GCDExtended()**) to find the private exponent **D**.
5. For each image:
   - Initialize an empty array **enc** to store the encrypted pixel values.
   - Iterate over the pixels of the image.
   - Encrypt each pixel value **(r, g, b)** using the RSA encryption formula: **C = (r^E mod N, g^E mod N, b^E mod N)**.
   - Reduce the encrypted pixel values modulo 256.
   - Update the image pixels with the encrypted values.
   - Display the encrypted image using **matplotlib.pyplot.imshow()** and **matplotlib.pyplot.show()**.
6. For each encrypted image:
   - Iterate over the pixels.
   - Decrypt each encrypted pixel value **(C1, C2, C3)** using the RSA decryption formula: **M = (C1^D mod N, C2^D mod N, C3^D mod N)**.
   - Update the image pixels with the decrypted values.
   - Display the decrypted image using **matplotlib.pyplot.imshow()** and **matplotlib.pyplot.show()**.

III. CODE SNIPPETS

```
#!/usr/bin/env python
# coding: utf-8

# <h1>Image Encryption and Decryption using RSA Algorithm
#     <br>Khushi Jain
#     <br>20BCE2664
#     <br> CODE SNIPPETS</h1>

# In[1]:


#Importing Libraries
import cv2
import numpy as np
import glob
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')


# In[2]:


#Reading Images
images = [cv2.imread(img) for img in glob.glob("Images/*.jpg")]
for img in images:
    imgArray = np.array(img)
    plt.imshow(imgArray)
    plt.show()


# <h2>RSA Algorithm</h2>

# <h3>I. Generate Two Large Prime Numbers (p, q) randomly</h3>

# In[3]:


from random import randrange, getrandbits

def Power(a, d, n):
    res = 1;
    while d != 0:
        if d%2 == 1:
            res = ((res%n) * (a%n))%n
        a = ((a%n) * (a%n))%n
        d >>= 1
    return res;

def mllRabin(N, d):
    a = randrange(2, N - 1)
    x = Power(a, d, N);
```

```
    if x == 1 or x == N - 1:
        return True;

    else:
        while(d != N - 1):
            x = ((x%N)*(x%N))%N;
            if x == 1:
                return False;

            if x == N - 1:
                return True;

            d <<= 1;
    return False;

def isPrime(N, K):
    if N == 3 or N == 2:
        return True;

    if N <= 1 or N%2 == 0:
        return False;

    #d should be such that d * (2 ^ r) = x - 1
    d = N - 1
    while d%2 != 0:
        d /= 2;

    for _ in range(K):
        if not mllRabin(N, d):
            return False;
    return True;

def genPrimeCand(len):
    #Generate Random Bits
    p = getrandbits(len)

    #Apply a mask to set MSB and LSB to 1
    #Set MSB = 1 to make sure we have a number of 1024 bits.
    #Set LSB = 1 to make sure we get an Odd Number.
    p |= (1 << len - 1) | 1
    return p

def genPrimeNum(len):
    A = 24
    while not isPrime(A, 128):
        A = genPrimeCand(len)
    return A

len = 5
# P = genPrimeNum(len)
# Q = genPrimeNum(len)
```

```
P = 19
Q = 17

print(P)
print(Q)


# <h3>II. Generate Two Large Prime Numbers (p, q) randomly</h3>

# In[4]:


N = P * Q
eulerTotient = (P - 1) * (Q - 1)
print(N)
print(eulerTotient)


# <h3>
# III. Find E such that GCD(E, eulerTotient) = 1 (e -> should be co-prime)
# such that it satifies Condition 1
# </h3>

# In[5]:


from math import gcd

def findPubExp(eulerTotient):
    for E in range(2, eulerTotient):
        if gcd(E, eulerTotient) == 1:
            return E
    return None

E = findPubExp(eulerTotient)
print(E)


# <h3>IV. Find D</h3>

# For Finding D: It must satisy this property -> (D * E)Mod(eulerTotient) = 1;
# #There are two choices:
# 1. Choose D randomly and check which condition is satisfying above condition
# 2. Finding D we can use Extended Euclidean Algorithm:
#    ax + by = 1 i.e eulerTotient (x) + E(y) = GCD(eulerTotient, e)
# Best Approach -> 2 Extended Euclidean Algorithm

# In[6]:


def GCDExtended(E, eulerTotient):
```

```
    a1, a2, b1, b2, d1, d2 = 1, 0, 0, 1, eulerTotient, E

    while d2 != 1:
        #k
        k = (d1//d2)

        #a
        tmp = a2
        a2 = a1 - (a2 * k)
        a1 = tmp

        #b
        tmp = b2
        b2 = b1 - (b2 * k)
        b1 = tmp

        #d
        tmp = d2
        d2 = d1 - (d2 * k)
        d1 = tmp

    D = b2

    if D > eulerTotient:
        D = D%eulerTotient
    elif D < 0:
        D = D + eulerTotient

    return D

D = GCDExtended(E, eulerTotient)
print(D)


# In[7]:


for img in images:
    r, c = img.shape[0], img.shape[1]
    enc = [
        [
            0 for x in range(3000)
        ]

        for y in range(3000)
    ]


# <h3>V. Encryption</h3>

# In[8]:
```

```
for img in images:
    enc = np.zeros_like(img)
    h, w, _ = img.shape

    for i in range(h):
        for j in range(w):
            r, g, b = img[i, j]
            C1 = Power(r, E, N)
            C2 = Power(g, E, N)
            C3 = Power(b, E, N)

            enc[i][j] = [C1, C2, C3]

            C1 = C1 % 256
            C2 = C2 % 256
            C3 = C3 % 256

            img[i, j] = [C1, C2, C3]

    plt.imshow(img)
    plt.show()


# <h3>VI. Decryption</h3>

# In[9]:


for img in images:
    h, w, _ = enc.shape

    for i in range(h):
        for j in range(w):
            r, g, b = enc[i][j]
            D1 = Power(r, D, N)
            D2 = Power(g, D, N)
            D3 = Power(b, D, N)

            img[i, j] = [D1, D2, D3]

    plt.imshow(img)
    plt.show()
```

IV. SECURITY ANALYSIS

1. Prime Number Generation: The code uses the genPrimeNum() function to generate two large prime numbers, P and Q. It employs the Miller-Rabin primality test with a parameter K for probabilistic primality checking. The security of RSA relies on the difficulty of factoring large composite numbers into their prime factors. If the prime number generation algorithm is secure and produces large, random primes, it enhances the security of the RSA algorithm.

2. Key Generation: The code generates the public key (N, E) and private key (N, D) based on the selected prime numbers P and Q. It ensures that the public exponent E is relatively prime to the Euler totient function eulerTotient. The private exponent D is calculated using the Extended Euclidean Algorithm. The security of RSA heavily depends on the secrecy of the private key. If the key generation process is implemented correctly, it helps maintain the confidentiality and integrity of the encrypted data.

3. Encryption: The code performs RSA encryption by raising each pixel value (r, g, b) to the power of E and taking the modulo N. The encrypted pixel values (C1, C2, C3) are reduced modulo 256. This encryption process provides confidentiality for the image data, as an attacker without the private key would have difficulty recovering the original pixel values from the encrypted ones. However, the security of the encryption relies on the strength of the chosen prime numbers and the randomness of the encryption process.

4. Decryption: The code performs RSA decryption by raising each encrypted pixel value (C1, C2, C3) to the power of D and taking the modulo N. The decrypted pixel values (M1, M2, M3) are recovered. If the private key is kept secret, only the authorized recipient with the private key can decrypt the encrypted image and obtain the original pixel values. The security of decryption relies on the secrecy of the private key and the inability of attackers to factorize the large composite number N.

5. Key Security: The security of the RSA algorithm depends on the strength and secrecy of the private key (N, D). It is crucial to keep the private key confidential and protect it from unauthorized access. If an attacker gains access to the private key, they can decrypt the encrypted images and compromise the security of the system.

6. Key Length: The code uses a variable len to specify the length of the prime numbers generated for the RSA key pair. The length chosen for the prime numbers affects the security of the RSA algorithm. Longer key lengths provide stronger security against brute-force attacks and factorization methods. It is recommended to use key lengths of at least 2048 bits for adequate security.

7. Randomness: The code uses random number generation for various purposes, such as generating prime candidates and selecting the public exponent E. The security of RSA relies on the unpredictability and randomness of these generated values. If the random number generation is not truly random or predictable, it could weaken the security of the RSA algorithm and make it vulnerable to attacks.

V. REFERENCES

[1] https://www.techtarget.com/searchsecurity/definition/asymmetric-cryptography
[2] https://www.tutorialspoint.com/symmetric-key-algorithms#
[3] https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm
[4] https://www.tutorialspoint.com/what-are-the-symmetric-key-cryptography-in-information-security
[5] https://www.educba.com/asymmetric-encryption/

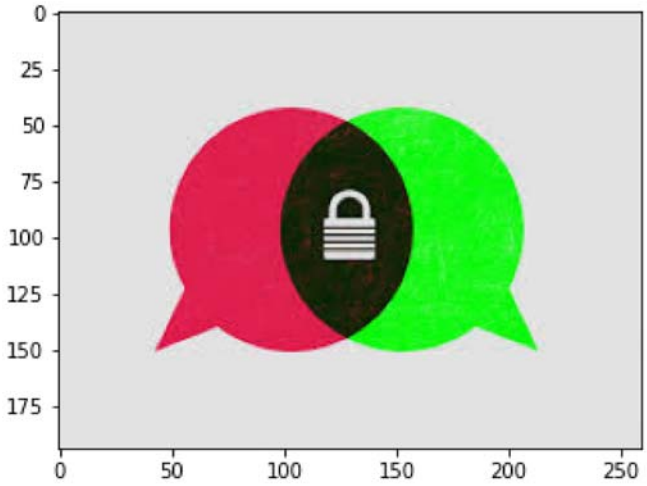
Image Encryption and Decryption using RSA Algorithm
Khushi Jain
20BCE2664
CODE SNIPPETS

In [1]:

```
#Importing Libraries
import cv2
import numpy as np
import glob
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:

```
#Reading Images
images = [cv2.imread(img) for img in glob.glob("Images/*.jpg")]
for img in images:
    imgArray = np.array(img)
    plt.imshow(imgArray)
    plt.show()
```



RSA Algorithm

I. Generate Two Large Prime Numbers (p, q) randomly

In [3]:

```
from random import randrange, getrandbits

def Power(a, d, n):
    res = 1;
    while d != 0:
        if d%2 == 1:
            res = ((res%n) * (a%n))%n
        a = ((a%n) * (a%n))%n
        d >>= 1
    return res;

def mllRabin(N, d):
    a = randrange(2, N - 1)
    x = Power(a, d, N);
    if x == 1 or x == N - 1:
        return True;

    else:
        while(d != N - 1):
            x = ((x%N)*(x%N))%N;
            if x == 1:
                return False;

            if x == N - 1:
                return True;

            d <<= 1;
    return False;

def isPrime(N, K):
    if N == 3 or N == 2:
        return True;

    if N <= 1 or N%2 == 0:
        return False;

    #d should be such that d * (2 ^ r) = x - 1
    d = N - 1
    while d%2 != 0:
        d /= 2;

    for _ in range(K):
        if not mllRabin(N, d):
            return False;
    return True;

def genPrimeCand(len):
    #Generate Random Bits
    p = getrandbits(len)

    #Apply a mask to set MSB and LSB to 1
    #Set MSB = 1 to make sure we have a number of 1024 bits.
    #Set LSB = 1 to make sure we get an Odd Number.
    p |= (1 << len - 1) | 1
    return p

def genPrimeNum(len):
    A = 24
    while not isPrime(A, 128):
```

```
        A = genPrimeCand(len)
    return A

len = 5
# P = genPrimeNum(len)
# Q = genPrimeNum(len)
P = 19
Q = 17

print(P)
print(Q)
```

```
19
17
```

II. Generate Two Large Prime Numbers (p, q) randomly

In [4]:

```
N = P * Q
eulerTotient = (P - 1) * (Q - 1)
print(N)
print(eulerTotient)
```

```
323
288
```

III. Find E such that GCD(E, eulerTotient) = 1 (e -> should be co-prime) such that it satifies Condition 1

In [5]:

```
from math import gcd

def findPubExp(eulerTotient):
    for E in range(2, eulerTotient):
        if gcd(E, eulerTotient) == 1:
            return E
    return None

E = findPubExp(eulerTotient)
print(E)
```

```
5
```

IV. Find D

For Finding D: It must satisy this property -> (D * E)Mod(eulerTotient) = 1; #There are two choices:

1. Choose D randomly and check which condition is satisfying above condition
2. Finding D we can use Extended Euclidean Algorithm: ax + by = 1 i.e eulerTotient (x) + E(y) = GCD(eulerTotient, e) Best Approach -> 2 Extended Euclidean Algorithm

In [6]:

```
def GCDExtended(E, eulerTotient):
    a1, a2, b1, b2, d1, d2 = 1, 0, 0, 1, eulerTotient, E

    while d2 != 1:
        #k
        k = (d1//d2)

        #a
        tmp = a2
        a2 = a1 - (a2 * k)
        a1 = tmp

        #b
        tmp = b2
        b2 = b1 - (b2 * k)
        b1 = tmp

        #d
        tmp = d2
        d2 = d1 - (d2 * k)
        d1 = tmp

    D = b2

    if D > eulerTotient:
        D = D%eulerTotient
    elif D < 0:
        D = D + eulerTotient

    return D

D = GCDExtended(E, eulerTotient)
print(D)
```

```
173
```

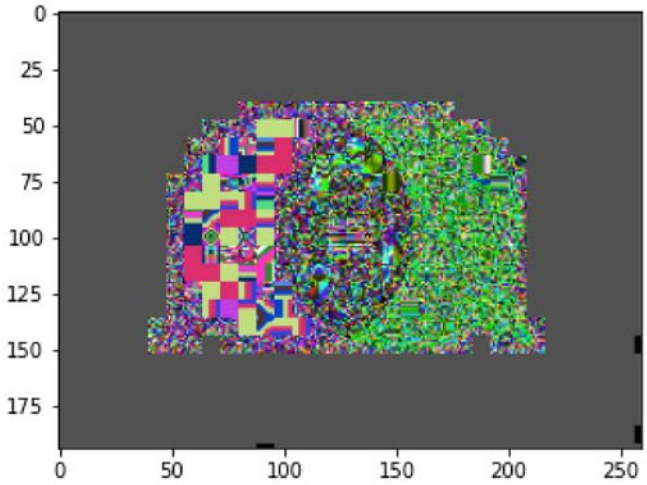
In [7]:

```
for img in images:
    r, c = img.shape[0], img.shape[1]
    enc = [
        [
            0 for x in range(3000)
        ]

        for y in range(3000)
    ]
```

V. Encryption

In [8]:

```
for img in images:
    enc = np.zeros_like(img)
    h, w, _ = img.shape

    for i in range(h):
        for j in range(w):
            r, g, b = img[i, j]
            C1 = Power(r, E, N)
            C2 = Power(g, E, N)
            C3 = Power(b, E, N)

            enc[i][j] = [C1, C2, C3]

            C1 = C1 % 256
            C2 = C2 % 256
            C3 = C3 % 256

            img[i, j] = [C1, C2, C3]

    plt.imshow(img)
    plt.show()
```



VI. Decryption

In [9]:

```
for img in images:
    h, w, _ = enc.shape

    for i in range(h):
        for j in range(w):
            r, g, b = enc[i][j]
            D1 = Power(r, D, N)
            D2 = Power(g, D, N)
            D3 = Power(b, D, N)

            img[i, j] = [D1, D2, D3]

    plt.imshow(img)
    plt.show()
```