# NAME-SHREYA DOKANIA

# Symmetric key algorithm: AES

**Briefly explain how the algorithm works**

AES (Advanced Encryption Standard) is a symmetric key algorithm that operates on fixed-size blocks of data. It works by applying a series of substitution and permutation operations to the input data using a round-based encryption process.

1. Key Expansion: The input secret key is expanded into a set of round keys. The number of round keys depends on the key length used (128, 192, or 256 bits).
2. Initial Round: The input data (plaintext) is divided into blocks, typically 128 bits. The first round begins with an AddRoundKey operation, where each byte of the block is XORed with a corresponding byte from the round key.
3. Rounds: AES employs a fixed number of rounds, which varies depending on the key length. Each round consists of four operations applied to the data block: SubBytes, ShiftRows, MixColumns, and AddRoundKey. These operations introduce confusion and diffusion, enhancing the encryption's strength.
   a. SubBytes: Each byte of the block is replaced with a corresponding byte from an S-Box lookup table. This substitution step adds non-linearity to the algorithm.
   b. ShiftRows: Bytes in each row of the block are cyclically shifted to the left. This operation provides diffusion, ensuring that changes in one byte affect multiple output bytes.
   c. MixColumns: Each column of the block is transformed by combining and manipulating the four bytes using matrix multiplication. This step further enhances the diffusion properties.
   d. AddRoundKey: The current round key is XORed with the block. Each byte in the block is combined with the corresponding byte from the round key.
4. Final Round: The final round excludes the MixColumns operation, retaining SubBytes, ShiftRows, and AddRoundKey.
5. Output: After the final round, the resulting block is the encrypted ciphertext.

To decrypt the ciphertext, the inverse operations of AES are applied using the same round keys in reverse order. This process effectively reverses the encryption steps and recovers the original plaintext.

AES is widely recognized for its security and efficiency, making it the most commonly used symmetric key algorithm in various applications.

## **Discuss the key strengths and advantages of the algorithm.**

AES (Advanced Encryption Standard) has several key strengths and advantages that contribute to its widespread adoption and recognition as a robust encryption algorithm. Here are some of its key strengths:

1. Security: AES has been extensively analyzed and reviewed by cryptographic experts worldwide. It has withstood rigorous scrutiny and evaluation, which has led to a high level of confidence in its security. AES has no known practical vulnerabilities when used correctly and with sufficient key lengths.
2. Efficiency: AES is highly efficient and computationally fast, making it suitable for a wide range of applications. It is designed to be implemented in both hardware and software environments efficiently. AES performs encryption and decryption operations relatively quickly, making it suitable for real-time applications such as secure communication protocols and disk encryption.
3. Flexibility: AES supports three key lengths: 128, 192, and 256 bits. The ability to choose different key lengths allows for a balance between security and performance, depending on the specific requirements of the application. AES can be tailored to fit the desired level of security while accommodating various computing platforms and resource constraints.
4. Wide Adoption: AES has gained worldwide acceptance and is widely adopted as the de facto standard for symmetric encryption. Its broad acceptance ensures interoperability between different systems and platforms. AES is implemented in numerous hardware and software products, including cryptographic libraries, operating systems, network devices, and secure communication protocols.
5. Resistance to Attacks: AES is designed to resist various cryptographic attacks, including brute-force attacks, differential and linear cryptanalysis, and known-plaintext attacks. AES employs complex mathematical operations and substitution-permutation layers to provide strong resistance against known attack methods.
6. Standardization and Trust: AES is an open standard published by the National Institute of Standards and Technology (NIST) in the United States. Its transparent design, public scrutiny, and standardization contribute to the trust and reliability of the algorithm. The involvement of multiple cryptographic experts in the design and evaluation process enhances its credibility.

Overall, AES offers a combination of security, efficiency, flexibility, and widespread adoption that makes it a highly reliable and trusted encryption algorithm for protecting sensitive data in various applications.

**Identify any known vulnerabilities or weaknesses.**

While AES (Advanced Encryption Standard) is considered a highly secure encryption algorithm, there are a few vulnerabilities and weaknesses that have been identified over time. It's important to note that these weaknesses are relatively minor and require specific conditions or implementation flaws to be exploitable. Here are a couple of known vulnerabilities or weaknesses of AES:

1. Side-Channel Attacks: AES implementations can be susceptible to side-channel attacks, where an attacker gathers information about the encryption process by monitoring physical signals, such as power consumption, electromagnetic radiation, or timing measurements. By analyzing these side-channel leaks, an attacker may be able to extract the secret key. Implementations need to incorporate countermeasures, such as randomizing intermediate values or using constant-time algorithms, to mitigate these attacks.
2. Key Management: The security of any encryption system, including AES, heavily relies on proper key management. Weaknesses in key generation, storage, distribution, or handling can undermine the security of AES. For example, if a weak or predictable key is used, it could be vulnerable to brute-force attacks. Additionally, if the key is compromised or mishandled during key exchange, encryption strength can be compromised.

It's important to note that these vulnerabilities do not stem from weaknesses in the AES algorithm itself but rather from implementation flaws or the broader context of the encryption system. When properly implemented and used with secure key management practices, AES is considered highly secure and resistant to attacks

**Provide real-world examples of where the algorithm is commonly used.**

AES (Advanced Encryption Standard) is widely used in various real-world applications to secure sensitive data and ensure confidentiality. Here are some examples of where the AES algorithm is commonly employed:

1. Secure Communication: AES is extensively used to secure communications over networks. It forms the foundation of secure protocols such as Transport Layer Security (TLS) and Secure Shell (SSH), which are employed for secure web browsing, email communication, remote access to servers, and secure file transfer.
2. Disk and File Encryption: AES is utilized for encrypting data stored on hard drives, solid-state drives (SSDs), and other storage media. Full disk encryption solutions like

BitLocker (Windows) and FileVault (Mac) use AES to protect the entire disk contents. AES is also employed in file encryption software to encrypt individual files or folders.

3. Virtual Private Networks (VPNs): AES is a fundamental component of VPNs, which create secure and private connections over public networks. VPN protocols such as OpenVPN and IPsec use AES to encrypt the VPN traffic, ensuring confidentiality and integrity of the transmitted data.
4. Wireless Networks: AES is employed in securing wireless networks, particularly in Wi-Fi networks using the WPA2 (Wi-Fi Protected Access II) standard. AES-CCMP (Counter Mode with Cipher Block Chaining Message Authentication Code Protocol) is the encryption algorithm used in WPA2 to protect wireless communications.
5. Secure Messaging and Voice Calls: Many messaging and voice call applications utilize AES for end-to-end encryption, ensuring that only the intended recipients can access the messages or calls. Examples include popular messaging apps like Signal, WhatsApp (when using end-to-end encryption), and secure voice call services.
6. Database Encryption: AES is often employed in database systems to encrypt sensitive data at rest. It helps protect sensitive information stored in databases, providing an additional layer of security to prevent unauthorized access.
7. Financial Transactions: AES is commonly used in securing financial transactions, including online banking, e-commerce, and mobile payment systems. It ensures the confidentiality and integrity of sensitive financial data transmitted over networks.

These examples highlight the widespread adoption of AES across different domains to safeguard data and ensure secure communication and storage.

# Asymmetric key algorithm: RSA

## Briefly explain how the algorithm works

The RSA (Rivest-Shamir-Adleman) algorithm is an asymmetric encryption algorithm that involves the use of a public key and a private key. Here's a brief explanation of how RSA works:

Key Generation:

1. Choose two large prime numbers, p and q.
2. Calculate the modulus, N, by multiplying p and q (N = p * q).
3. Calculate Euler's totient function, φ(N), which is the number of positive integers less than N that are coprime with N (φ(N) = (p-1) * (q-1)).
4. Choose an integer e (1 < e < φ(N)) that is coprime with φ(N) and forms the public key (N, e).

5. Compute the modular multiplicative inverse of e modulo φ(N) to find d, which forms the private key (N, d).

Encryption:

1. To encrypt a plaintext message, convert it to a numerical representation (usually using a predetermined mapping).
2. The recipient's public key (N, e) is used. Apply the encryption formula: ciphertext = (plaintext^e) mod N.
3. The resulting ciphertext is the encrypted message.

Decryption:

1. To decrypt the ciphertext, the recipient uses their private key (N, d).
2. Apply the decryption formula: plaintext = (ciphertext^d) mod N.
3. The resulting plaintext is the decrypted message.

Key Distribution and Digital Signatures:

RSA is also used for key distribution and digital signatures. In key distribution, the sender encrypts a symmetric encryption key with the recipient's public key, allowing secure communication using symmetric encryption after the initial key exchange. In digital signatures, the sender uses their private key to sign a message, and the recipient can verify the signature using the sender's public key, ensuring the authenticity and integrity of the message.

The security of RSA is based on the difficulty of factoring large numbers. The strength of RSA relies on using sufficiently large prime numbers for key generation and the protection of the private key.

It's worth noting that RSA encryption and decryption operations can be computationally expensive, especially for large key sizes. Therefore, RSA is often used in conjunction with symmetric encryption algorithms like AES, where RSA is used for key distribution and encryption of the symmetric key, while the actual data is encrypted using the symmetric algorithm for efficiency.

## **Discuss the key strengths and advantages of the algorithm.**

The RSA (Rivest-Shamir-Adleman) algorithm has several key strengths and advantages that have contributed to its widespread use as an asymmetric encryption algorithm. Here are some of its key strengths:

1. Key Exchange: RSA facilitates secure key exchange between parties who have not previously shared a secret key. The use of public and private key pairs allows for secure communication without the need for a prior shared secret.
2. Encryption and Decryption: RSA can be used for both encryption and decryption. The recipient's public key is used for encryption, ensuring that only the corresponding private key can decrypt the ciphertext. This property enables secure communication and confidentiality.
3. Digital Signatures: RSA enables the creation and verification of digital signatures. By signing a message with the private key, the sender can provide proof of authenticity and integrity. The recipient can verify the signature using the sender's public key, ensuring that the message has not been tampered with during transit.
4. Security based on Mathematical Difficulty: The security of RSA is based on the difficulty of factoring large composite numbers into their prime factors. Breaking RSA requires factoring the modulus, which becomes computationally infeasible for sufficiently large prime numbers used in the key generation process. As of now, no efficient algorithm exists to factor large numbers, making RSA a secure choice.
5. Widely Supported and Interoperable: RSA is widely supported and implemented in various cryptographic libraries, operating systems, and applications. This widespread adoption ensures interoperability between different systems and platforms, enabling secure communication across a wide range of devices and networks.
6. Scalability: RSA can handle varying key sizes, making it scalable to meet different security requirements. Larger key sizes provide stronger security but may require more computational resources. The flexibility to choose appropriate key sizes allows for a balance between security and performance.
7. Proven and Well-Studied: RSA has been extensively studied, analyzed, and scrutinized by the cryptographic community over several decades. Its security and strength have been tested through rigorous examination and peer review, increasing confidence in its reliability and effectiveness.

Despite its strengths, RSA does have some limitations. It is computationally intensive, especially for large key sizes, which can impact performance in resource-constrained environments. Additionally, RSA is vulnerable to attacks if implemented incorrectly or if the key management practices are weak.

Overall, RSA's key strengths include its ability to facilitate secure key exchange, support encryption and decryption, provide digital signatures, rely on mathematical difficulty for security, wide support and interoperability, scalability, and extensive research and analysis. These attributes have made RSA a widely used and trusted asymmetric encryption algorithm in various applications.

**<u>Identify any known vulnerabilities or weaknesses.</u>**

While RSA (Rivest-Shamir-Adleman) is a widely used asymmetric encryption algorithm, it is not without vulnerabilities and weaknesses. Here are a few known vulnerabilities or weaknesses of RSA:

1. Key Length: The security of RSA depends on the length of the keys used. With advancements in computing power and mathematical algorithms, the recommended key lengths have increased over time. Using shorter key lengths can make RSA susceptible to attacks, including factorization methods such as the General Number Field Sieve (GNFS) or the Quadratic Sieve (QS). It is crucial to use key lengths that are currently considered secure, such as 2048 bits or higher.
2. Implementation Flaws: Vulnerabilities can arise from flaws in the implementation of RSA in software or hardware. These flaws may include poor random number generation, insufficient padding schemes, side-channel attacks, or timing attacks. Implementations need to follow recommended standards and best practices to ensure the security of RSA.
3. Timing Attacks: Timing attacks exploit the variations in execution time of cryptographic operations to gain information about the private key. By analyzing the time taken for different operations, an attacker can potentially extract the private key. Countermeasures such as constant-time implementations or blinding techniques are necessary to mitigate timing attacks.
4. Side-Channel Attacks: Side-channel attacks involve analyzing unintended information leaks such as power consumption, electromagnetic radiation, or acoustic emanations during cryptographic operations. These side-channel leaks can reveal sensitive information, including the private key. Countermeasures, such as implementing techniques like masking or power analysis resistance, are required to protect against side-channel attacks.
5. Quantum Computing Threat: RSA's security relies on the computational difficulty of factoring large numbers and the discrete logarithm problem. Quantum computers have the potential to solve these problems efficiently using Shor's algorithm. As quantum computing advances, it poses a threat to the security of RSA and other public-key cryptosystems. Post-quantum cryptography algorithms are being developed as potential replacements for RSA to ensure secure communication in a post-quantum era.

It's important to note that these vulnerabilities and weaknesses primarily stem from specific implementation flaws, insufficient key lengths, or the potential future advancements in computing technologies. When implemented correctly with appropriate key lengths and secure practices, RSA remains a widely used and secure encryption algorithm. Regular updates and adherence to best practices are crucial to addressing any emerging vulnerabilities and maintaining the security of RSA.

## **Provide real-world examples of where the algorithm is commonly used.**

The RSA (Rivest-Shamir-Adleman) algorithm is widely used in various real-world applications that require secure communication, authentication, and data protection. Here are some examples of where the RSA algorithm is commonly employed:

1. Secure Email: RSA is often used in email encryption protocols like Pretty Good Privacy (PGP) and S/MIME (Secure/Multipurpose Internet Mail Extensions). RSA ensures the confidentiality and integrity of email messages by encrypting the content or digitally signing the messages using the sender's private key.
2. SSL/TLS Encryption: RSA is an integral part of the SSL/TLS (Secure Sockets Layer/Transport Layer Security) protocols, which provide secure communication over the internet. RSA is used during the SSL/TLS handshake to establish a secure connection and negotiate a shared symmetric key for encrypting the data exchange.
3. Secure Web Browsing: RSA is employed in HTTPS (HTTP Secure), the secure version of the HTTP protocol used for secure web browsing. RSA is used during the SSL/TLS handshake to authenticate the server, establish a secure connection, and encrypt the data transmitted between the web browser and the server.
4. Virtual Private Networks (VPNs): RSA plays a role in secure VPN connections. It is used for key exchange and authentication in VPN protocols like OpenVPN and IPsec. RSA ensures the secure establishment of the VPN tunnel and the encryption of the VPN traffic.
5. Digital Signatures: RSA is widely used for digital signatures, which provide a means of verifying the authenticity and integrity of digital documents, software, and code. RSA signatures are used in applications like software updates, digital certificates, code signing, and secure document verification.
6. Secure Authentication: RSA is employed in authentication protocols such as Secure Shell (SSH) for secure remote logins and secure file transfers. RSA-based authentication allows users to authenticate themselves securely to remote systems using public and private key pairs.
7. Smart Cards and Secure Tokens: RSA is often used in smart card technology and secure tokens for authentication, secure storage of sensitive information, and secure transactions. RSA provides a secure mechanism for generating and verifying digital signatures and encryption operations within these devices.
8. Secure File Storage: RSA encryption is used in various secure file storage and backup solutions. It ensures the confidentiality of data stored in encrypted form, protecting sensitive information from unauthorized access.

These examples demonstrate the widespread adoption and versatile applications of RSA in securing communication, ensuring data integrity, and providing authentication mechanisms in various domains.

# Hash Function: MD5

## Briefly explain how the algorithm works

The MD5 (Message Digest Algorithm 5) is a widely used cryptographic hash function that takes an input message and produces a fixed-size hash value, typically 128 bits (16 bytes). Here's a brief explanation of how the MD5 algorithm works:

1. Padding: The input message is padded to make its length a multiple of 512 bits. The padding is done in such a way that it includes a 1 bit followed by a series of 0 bits, and then the length of the original message is appended in binary format.
2. Initialization: MD5 initializes a 128-bit state (four 32-bit words) as prescribed by the algorithm. This state is used to keep track of intermediate hash values during the computation.
3. Processing: The padded message is divided into blocks of 512 bits. Each block goes through a series of rounds, where various bitwise operations, logical functions, and modular addition operations are performed on the state and the block to update the state.
4. Finalization: After processing all the blocks, the final state is transformed into the 128-bit hash value. The resulting hash value is typically represented as a sequence of 32 hexadecimal digits.

The MD5 algorithm is deterministic, meaning that the same input message will always produce the same hash value. It is designed to be fast and efficient, making it useful for applications like checksumming, fingerprinting, and verifying the integrity of files or data.

However, it's important to note that MD5 is considered insecure for cryptographic purposes due to its vulnerabilities. Researchers have discovered collision attacks, where two different messages can produce the same hash value, allowing for potential manipulation or forgery of data. Consequently, MD5 is no longer recommended for cryptographic security, and stronger hash functions like SHA-256 are preferred for secure applications.

## Discuss the key strengths and advantages of the algorithm.

While the MD5 (Message Digest Algorithm 5) algorithm was widely used in the past, it is now considered weak for cryptographic purposes due to its vulnerabilities. However, it still has some advantages and strengths in certain non-cryptographic applications. Here are a few key strengths and advantages of MD5:

1. Fast Computation: MD5 is a relatively fast hashing algorithm compared to some of its successors like SHA-256. Its efficient computation makes it suitable for applications that require quick processing of large amounts of data.
2. Wide Compatibility: MD5 has been widely implemented and supported in various systems, programming languages, and frameworks. It has become a standard hash function in many applications and libraries, making it compatible with a broad range of platforms and systems.
3. Data Integrity Verification: MD5 can be useful for verifying the integrity of data, such as files or messages, in situations where security is not a primary concern. By comparing the MD5 hash of a file before and after transmission or storage, one can quickly determine if any changes or corruption have occurred.
4. Non-Cryptographic Applications: MD5 can still be applied in non-cryptographic scenarios where the risk of malicious attacks is low. For example, MD5 can be used for checksumming, fingerprinting, or indexing purposes, where its speed and compatibility are advantageous.

It's important to note that the key strengths mentioned above are specific to non-cryptographic applications. For cryptographic purposes, such as data security, password storage, or digital signatures, MD5 is no longer recommended due to its vulnerabilities to collision attacks and the availability of more secure alternatives like SHA-256 or SHA-3.

To ensure security and integrity in cryptographic applications, it is strongly advised to use modern and secure hash functions that have been extensively analyzed and tested for robustness against known attacks.

## **Identify any known vulnerabilities or weaknesses.**

The MD5 (Message Digest Algorithm 5) algorithm has several known vulnerabilities and weaknesses that make it unsuitable for cryptographic applications. Here are the key vulnerabilities associated with MD5:

1. Collision Attacks: MD5 is vulnerable to collision attacks, where two different input messages can produce the same hash value. Researchers have demonstrated practical collision attacks on MD5, allowing attackers to create different messages with the same MD5 hash. This vulnerability poses a significant security risk as it enables potential data manipulation and forgery.
2. Preimage Attacks: MD5 is also susceptible to preimage attacks, where an attacker can find a different input message that produces a specific desired hash value. These attacks

undermine the integrity and non-repudiation properties of MD5, as an attacker can create a message with a known hash, potentially leading to unauthorized access or tampering.

3. Speed of Computation: While the fast computation speed of MD5 can be considered a strength in certain non-cryptographic applications, it also makes it more susceptible to brute-force attacks. The speed of modern computers and dedicated hardware significantly reduces the time required to search for a valid input producing a specific MD5 hash.
4. Weaknesses in Hash Function Design: MD5 employs a relatively simple design compared to more modern and secure hash functions. It relies on bitwise logical operations, modular addition, and a particular hash round structure. These design choices have been found to have vulnerabilities when subjected to cryptanalysis.

Due to these vulnerabilities, MD5 is no longer considered secure for cryptographic purposes. Its use in security-critical applications such as password storage, digital signatures, or data integrity verification is strongly discouraged. More secure alternatives, such as SHA-256 or SHA-3, are recommended for cryptographic hashing needs to ensure the integrity, authenticity, and confidentiality of data.

## **Provide real-world examples of where the algorithm is commonly used.**

While the MD5 (Message Digest Algorithm 5) algorithm is no longer considered secure for cryptographic purposes, it has historically been used in various real-world applications. However, its usage has diminished due to its vulnerabilities. Here are a few examples of where MD5 was commonly used in the past:

1. Checksums: MD5 checksums were widely used to verify the integrity of files or data. Software developers and users would calculate the MD5 hash of a file and compare it with the provided checksum to ensure that the file had not been altered or corrupted during transmission or storage.
2. Password Storage: MD5 was commonly used in older systems and applications to store user passwords. Instead of storing plaintext passwords, MD5 hashes of passwords were stored in databases. However, MD5 is no longer considered secure for password storage due to its vulnerability to preimage and collision attacks. Stronger password hashing algorithms like bcrypt, scrypt, or Argon2 are now recommended.
3. Data Deduplication: In certain data storage systems, MD5 hashes were used for deduplication, where identical data blocks are identified and stored only once to save space. MD5 hashes were calculated for data blocks, and if a hash match was found, the duplicate block was eliminated.
4. Non-Security Applications: MD5 has been used in non-security applications, such as checksumming for error detection, indexing and searching, or generating unique identifiers for data records.

It's important to note that these examples highlight the historical usage of MD5 and its relevance in non-cryptographic scenarios. However, due to its vulnerabilities to collision attacks and preimage attacks, MD5 is no longer recommended for security-critical applications. More secure hash functions like SHA-256 or SHA-3 are now widely used to ensure data integrity, authentication, and confidentiality in modern systems and applications.

## Implementation: AES

**CODE:**

```
from Crypto import Random

from Crypto.Cipher import AES

import os

import os.path

from os import listdir

from os.path import isfile, join

import time




class Encryptor:

   def __init__(self, key):

       self.key = key



   def pad(self, s):

       return s + b"\0" * (AES.block_size - len(s) % AES.block_size)
```

```
def encrypt(self, message, key, key_size=256):

    message = self.pad(message)

    iv = Random.new().read(AES.block_size)

    cipher = AES.new(key, AES.MODE_CBC, iv)

    return iv + cipher.encrypt(message)



def encrypt_file(self, file_name):

    with open(file_name, 'rb') as fo:

        plaintext = fo.read()

    enc = self.encrypt(plaintext, self.key)

    with open(file_name + ".enc", 'wb') as fo:

        fo.write(enc)

    os.remove(file_name)



def decrypt(self, ciphertext, key):

    iv = ciphertext[:AES.block_size]

    cipher = AES.new(key, AES.MODE_CBC, iv)

    plaintext = cipher.decrypt(ciphertext[AES.block_size:])

    return plaintext.rstrip(b"\0")



def decrypt_file(self, file_name):

    with open(file_name, 'rb') as fo:
        ciphertext = fo.read()

    dec = self.decrypt(ciphertext, self.key)

    with open(file_name[:-4], 'wb') as fo:

        fo.write(dec)

    os.remove(file_name)



def getAllFiles(self):

    dir_path = os.path.dirname(os.path.realpath(__file__))

    dirs = []

    for dirName, subdirList, fileList in os.walk(dir_path):

        for fname in fileList:

            if (fname != 'script.py' and fname != 'data.txt.enc'):

                dirs.append(dirName + "\\" + fname)

    return dirs



def encrypt_all_files(self):

    dirs = self.getAllFiles()

    for file_name in dirs:

        self.encrypt_file(file_name)



def decrypt_all_files(self):

    dirs = self.getAllFiles()
    for file_name in dirs:

        self.decrypt_file(file_name)




key =
b'[EX\xc8\xd5\xbfI{\xa2$\x05(\xd5\x18\xbf\xc0\x85)\x10nc\x94\x02)j\xdf\xcb\xc4\x94\x9d
(\x9e'

enc = Encryptor(key)

clear = lambda: os.system('cls')



if os.path.isfile('data.txt.enc'):

while True:

   password = str(input("Enter password: "))

   enc.decrypt_file("data.txt.enc")

   p = ''

   with open("data.txt", "r") as f:

       p = f.readlines()

   if p[0] == password:

       enc.encrypt_file("data.txt")

       break



while True:

   clear()
```

```python
       choice = int(input(

           "1. Press '1' to encrypt file.\n2. Press '2' to decrypt file.\n3. Press '3' to Encrypt all files in the directory.\n4. Press '4' to decrypt all files in the directory.\n5. Press '5' to exit.\n"))

       clear()

       if choice == 1:

           enc.encrypt_file(str(input("Enter name of file to encrypt: ")))

       elif choice == 2:

           enc.decrypt_file(str(input("Enter name of file to decrypt: ")))

       elif choice == 3:

           enc.encrypt_all_files()

       elif choice == 4:

           enc.decrypt_all_files()

       elif choice == 5:

           exit()

       else:

           print("Please select a valid option!")



else:

   while True:

       clear()

       password = str(input("Setting up stuff. Enter a password that will be used for decryption: "))
```

```
    repassword = str(input("Confirm password: "))

    if password == repassword:

        break

    else:

        print("Passwords Mismatched!")

f = open("data.txt", "w+")

f.write(password)

f.close()

enc.encrypt_file("data.txt")

print("Please restart the program to complete the setup")

time.sleep(15)
```

**<u>CODE EXPLANATION:</u>**

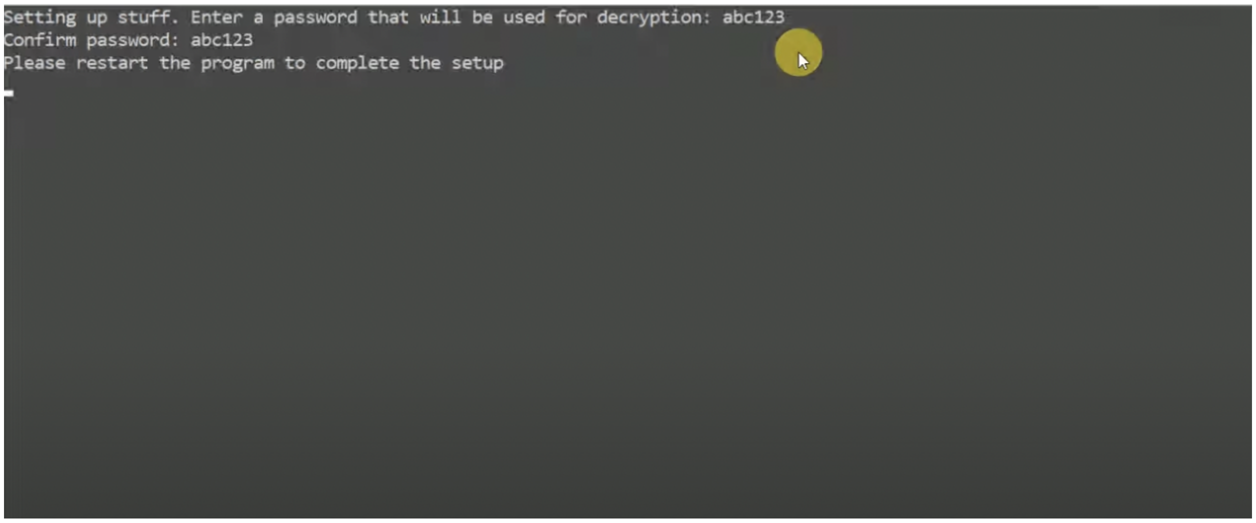
This code is an example of a file encryption program using the AES algorithm in Python. Let's break it down step by step:

1. Import necessary modules: The code imports the Random module from the Crypto package for generating a random IV and the AES module for AES encryption and decryption. It also imports other required modules such as os and os.path for file operations and time for delaying program execution.

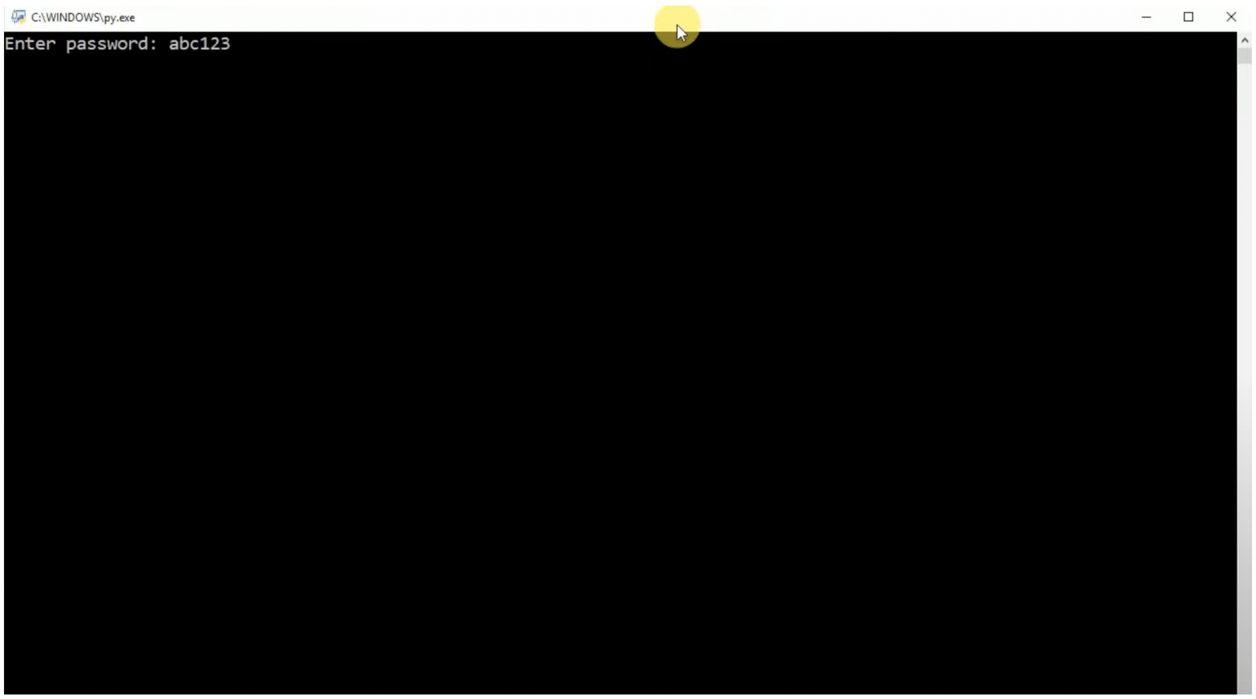
2. Define the Encryptor class: This class encapsulates all the encryption and decryption operations. It takes a key as input during initialization.
3. Implement padding: The pad method is used to pad the input message to a multiple of the AES block size.
4. Implement encryption: The encrypt method encrypts a message using AES with CBC mode. It generates a random IV, initializes an AES cipher with the key and IV, and returns the IV concatenated with the encrypted ciphertext.
5. Implement file encryption: The encrypt_file method reads a file, encrypts its contents using the encrypt method, and saves the encrypted data to a new file with the extension .enc.
6. Implement decryption: The decrypt method decrypts a ciphertext using AES with CBC mode. It separates the IV and ciphertext, initializes an AES cipher with the key and IV, and returns the decrypted plaintext.
7. Implement file decryption: The decrypt_file method reads an encrypted file, decrypts its contents using the decrypt method, and saves the decrypted data to a new file without the .enc extension.
8. Implement file enumeration: The getAllFiles method recursively traverses the current directory and returns a list of all file paths except the script file itself and the encrypted file.
9. Implement batch file encryption and decryption: The encrypt_all_files and decrypt_all_files methods iterate over the list of file paths obtained from getAllFiles and encrypt or decrypt each file using the corresponding methods.
10. Generate a random key and create an instance of Encryptor: A random key is generated and an instance of the Encryptor class is created, passing the key as an argument.
11. Check for the existence of an encrypted file: The code checks if an encrypted file called data.txt.enc exists.
12. Decrypt the file and prompt for password: If the encrypted file exists, the code prompts the user to enter a password for decryption. It then decrypts the file using the decrypt_file method and checks if the entered password matches the one stored in the file.
13. Perform encryption or decryption based on user input: If the password is correct, the code displays a menu with options for file encryption, file decryption, batch encryption, batch decryption, and program exit. Depending on the user's choice, it calls the corresponding methods to perform the desired encryption or decryption operation.
14. Setup password and encrypt it: If the encrypted file doesn't exist, the code guides the user through setting up a password. It writes the password to a file called data.txt, encrypts it using the encrypt_file method, and prompts the user to restart the program to complete the setup.

That's an overview of the provided code. It combines file encryption and decryption using AES with user input for password management and menu-based interaction.

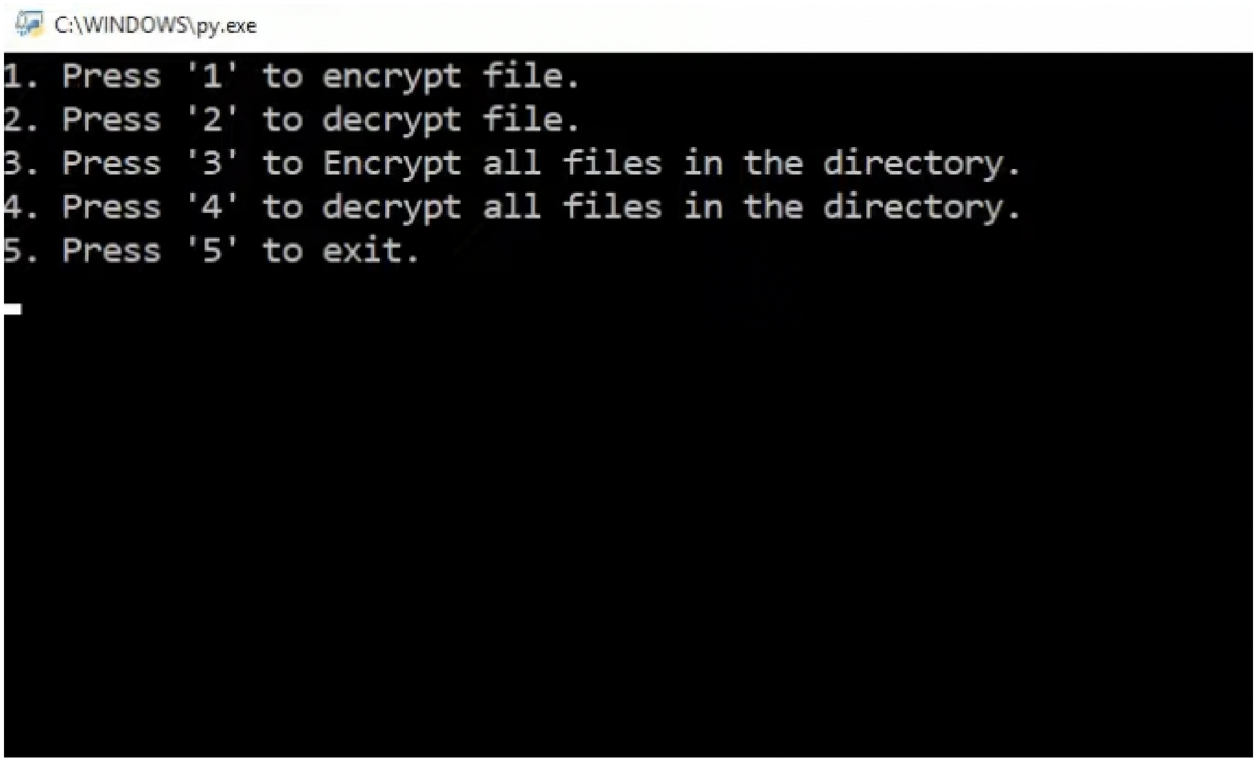
**OUTPUT:**



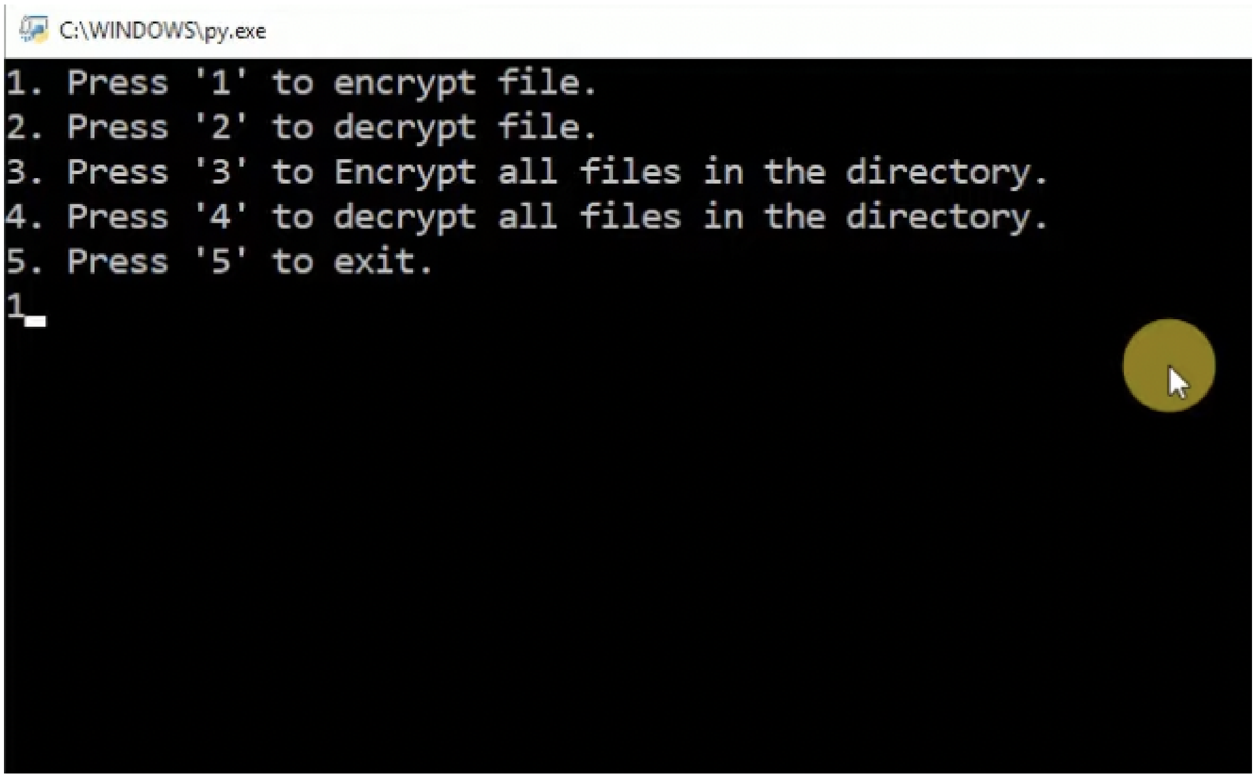
```
Setting up stuff. Enter a password that will be used for decryption: abc123
Confirm password: abc123
Please restart the program to complete the setup
```



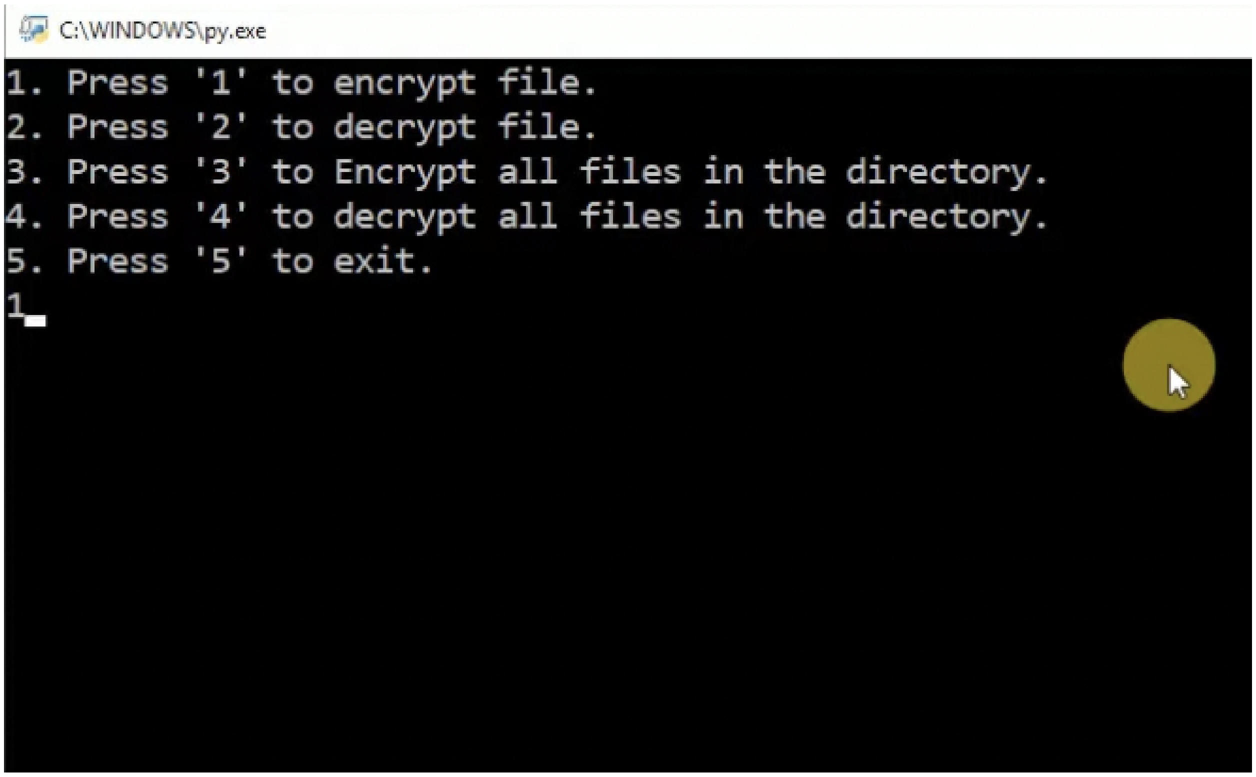
```
Enter password: abc123
```



```
1. Press '1' to encrypt file.
2. Press '2' to decrypt file.
3. Press '3' to Encrypt all files in the directory.
4. Press '4' to decrypt all files in the directory.
5. Press '5' to exit.
```

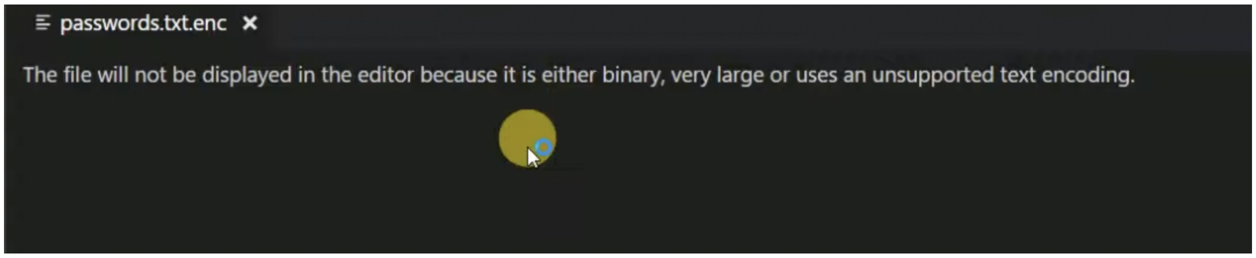


```
1. Press '1' to encrypt file.
2. Press '2' to decrypt file.
3. Press '3' to Encrypt all files in the directory.
4. Press '4' to decrypt all files in the directory.
5. Press '5' to exit.
1
```



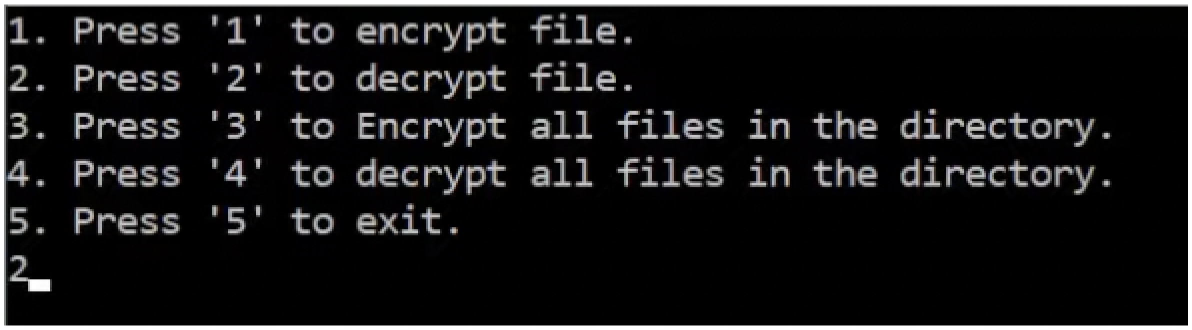
C:\WINDOWS\py.exe

```
1. Press '1' to encrypt file.
2. Press '2' to decrypt file.
3. Press '3' to Encrypt all files in the directory.
4. Press '4' to decrypt all files in the directory.
5. Press '5' to exit.
1
```

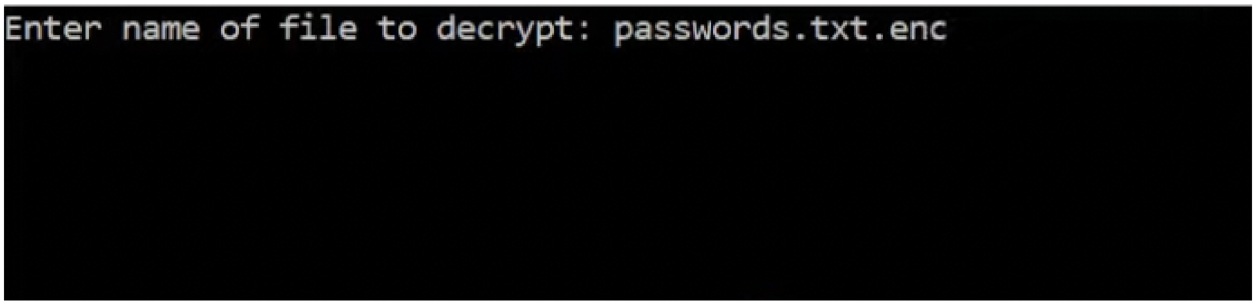


passwords.txt.enc

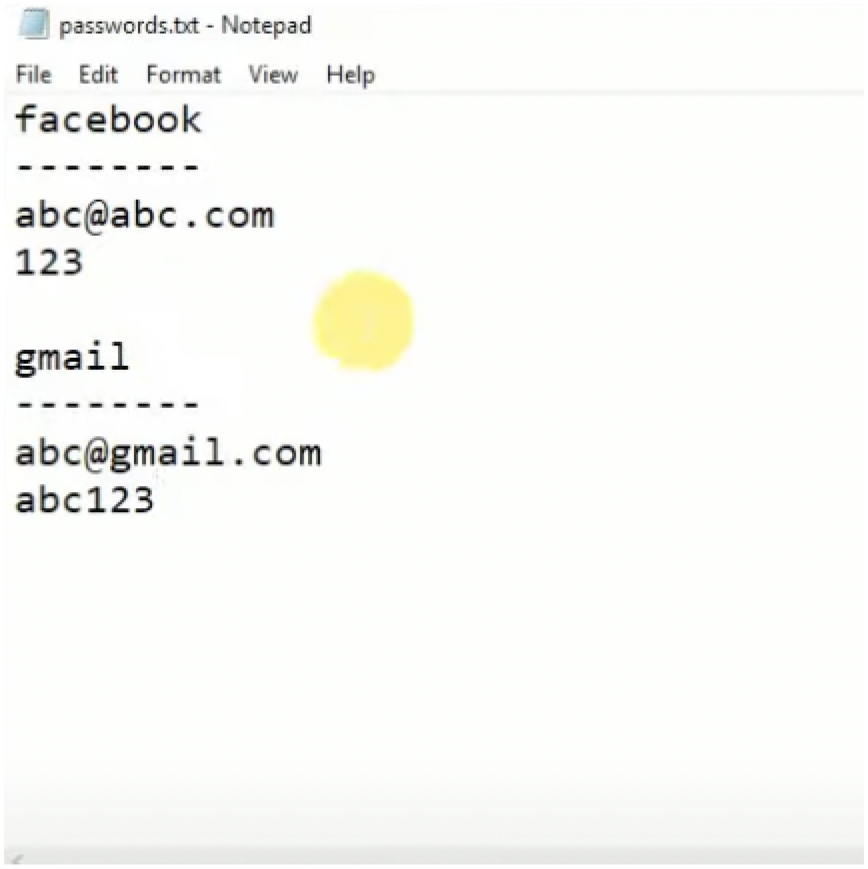
The file will not be displayed in the editor because it is either binary, very large or uses an unsupported text encoding.



```
1. Press '1' to encrypt file.
2. Press '2' to decrypt file.
3. Press '3' to Encrypt all files in the directory.
4. Press '4' to decrypt all files in the directory.
5. Press '5' to exit.
2
```



```
Enter name of file to decrypt: passwords.txt.enc
```



```
facebook
--------
abc@abc.com
123

gmail
--------
abc@gmail.com
abc123
```

**OUTPUT EXPLANATION:**

The output of the code will depend on the user's interactions with the program. Here's a breakdown of the possible scenarios and their corresponding outputs:

**If the data.txt.enc file exists:**

a. The program will prompt the user to enter a password.

b. If the entered password matches the one stored in the file, the program will display a menu with options for file encryption, file decryption, batch encryption, batch decryption, and program exit.

c. Based on the user's choice, the program will perform the selected operation and display appropriate messages indicating the success or failure of the operation.

**If the data.txt.enc file does not exist:**

a. The program will guide the user through setting up a password by entering and confirming it.

b. It will write the password to a file called data.txt and encrypt it.

c. The program will prompt the user to restart the program to complete the setup and then pause execution for 15 seconds.

d. After the 15-second pause, the program will terminate, and there will be no further output.

The specific output messages from the program will be displayed based on the user's inputs and the execution of the encryption or decryption operations.

## Implementation:RSA

**CODE:**

```
import random

import math



#select 2 large prime numbers

def generate_p_and_q():



   #Calculating 1 to 100 prime numbers
   numbs = [i for i in range(2,101)]


   for n in range(2,101):

       for i in range(2,math.ceil(n/2)+1):

           if n % i == 0:

               numbs.remove(n)

               break

           else:

               continue


   #Selecting any 2 prime numbers randomly


   p = random.choice(numbs)

   numbs.remove(p)

   q = random.choice(numbs)


   return p, q


p,q =  generate_p_and_q()


print(f'[+] p = {p} and q = {q}')
```

```
n = p * q

phi = (p - 1) * (q - 1)


print(f'[+] n = {n} and euler totient = {phi}')


#Calculating e -> gcd(e,phi) = 1 and 1 < e <phi.
def generate_e(phi):
   possible_e_values = []

   for i in range(2,phi):
       if math.gcd(i,phi) == 1:
           e=i
           possible_e_values.append(e)

   # print(possible_e_values)

   return random.choice(possible_e_values)
```

```
e = generate_e(phi)


print(f'[+] e = {e}')


def generate_d(e,phi):


   # d_list = []


   for i in range(2,phi):


       if (i*e) % phi == 1: #  ed mod(phi) = 1

           d = i # As every unique public key have only one unique private key.

           # d_list.append(d)

           break


   # print(d_list)

   return d


d = generate_d(e,phi)


print(f'[+] d = {d}')
```

```python
# Message should be less than n (msg < n)

msg = random.randint(1,n)

print(f'[+] msg : {msg}')

def encrypt(msg,e,n): #(msg^e) mod n
   c = pow(msg,e,n)
   return c

e_msg = encrypt(msg,e,n)

print(f'[+] Encrypted msg : {e_msg}')

def decrypt(msg,d,n): #(msg^d) mod n
   p = pow(msg,d,n)
   return p

d_msg = decrypt(e_msg,d,n)

print(f'[+] Decrypted msg : {d_msg}')
```

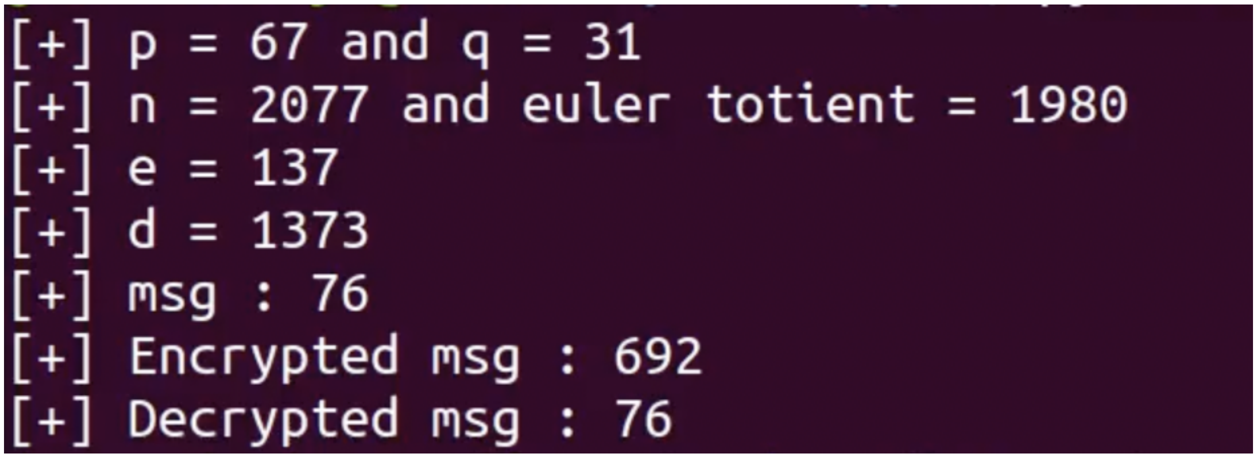
**CODE EXPLANATION:**

This code implements the generation of RSA key pairs and demonstrates the encryption and decryption process. Let's break it down step by step:

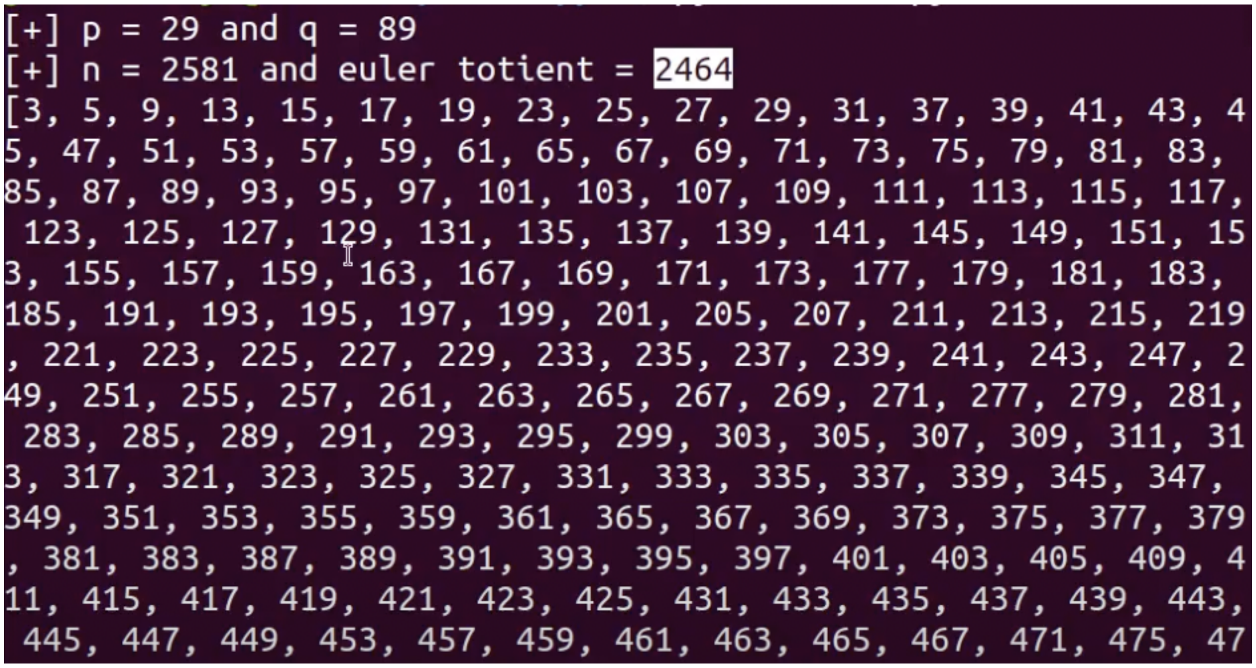
1. Import necessary modules: The code imports the random module for generating random numbers and the math module for mathematical operations.
2. Define the function generate_p_and_q: This function calculates the prime numbers between 2 and 100 and stores them in a list called numbs. It then selects two prime numbers randomly from the list and returns them as p and q.
3. Generate p and q values: The code calls the generate_p_and_q function and assigns the returned values to p and q.
4. Calculate n and Euler's totient function (phi): The code calculates the product of p and q as n. It also calculates Euler's totient function as (p - 1) * (q - 1) and assigns the result to phi.
5. Define the function generate_e: This function generates a suitable value of e that satisfies the condition gcd(e, phi) = 1 (i.e., e and phi are coprime). It iterates from 2 to phi and checks if i is coprime with phi. If it is, i is added to the list of possible e values. Finally, a random value is selected from the list and returned as e.
6. Generate e value: The code calls the generate_e function, passing phi as the argument, and assigns the returned value to e.
7. Define the function generate_d: This function generates a suitable value of d that satisfies the condition e * d mod phi = 1. It iterates from 2 to phi and checks if (i * e) mod phi is equal to 1. If it is, i is assigned to d. Since every unique public key has only one unique private key, the function breaks the loop after finding the first valid d value and returns it.
8. Generate d value: The code calls the generate_d function, passing e and phi as arguments, and assigns the returned value to d.
9. Generate a random message (msg): The code generates a random number between 1 and n and assigns it to msg.
10. Define the encrypt function: This function takes a message (msg), public key exponent (e), and modulus (n) as input. It uses the pow function to perform modular exponentiation of the message with e and n, and returns the ciphertext (c).
11. Encrypt the message: The code calls the encrypt function, passing msg, e, and n as arguments, and assigns the returned ciphertext to e_msg.
12. Define the decrypt function: This function takes a ciphertext (msg), private key exponent (d), and modulus (n) as input. It uses the pow function to perform modular exponentiation of the ciphertext with d and n, and returns the decrypted plaintext (p).
13. Decrypt the message: The code calls the decrypt function, passing e_msg, d, and n as arguments, and assigns the returned plaintext to d_msg.

14. Print the generated values and results: The code prints the generated values of p, q, n, phi, e, d, the original message

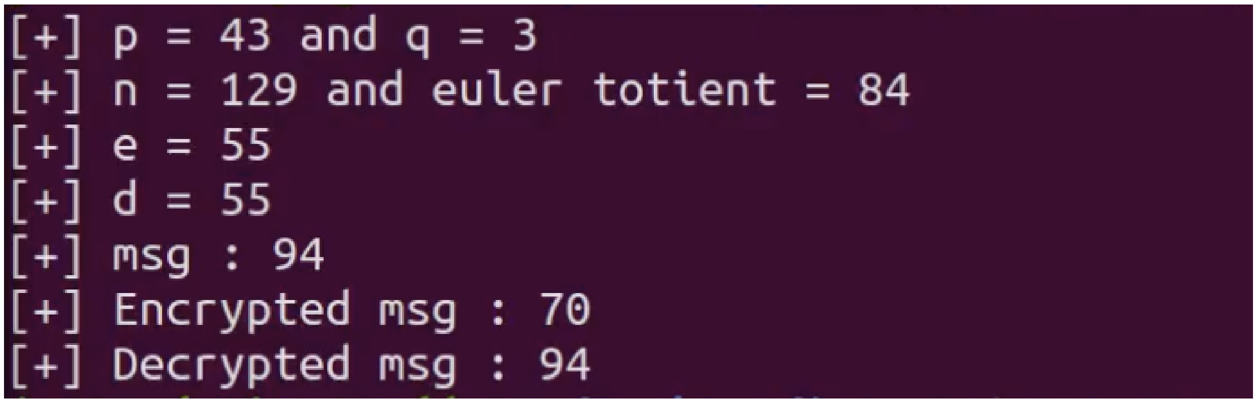
**OUTPUT:**



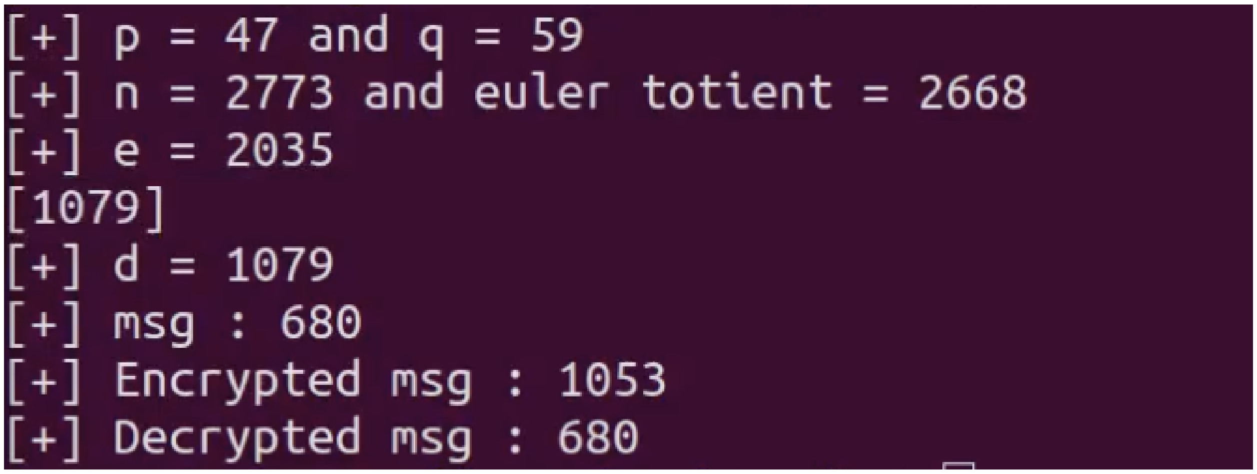
```
[+] p = 67 and q = 31
[+] n = 2077 and euler totient = 1980
[+] e = 137
[+] d = 1373
[+] msg : 76
[+] Encrypted msg : 692
[+] Decrypted msg : 76
```



```
[+] p = 29 and q = 89
[+] n = 2581 and euler totient = 2464
[3, 5, 9, 13, 15, 17, 19, 23, 25, 27, 29, 31, 37, 39, 41, 43, 4
5, 47, 51, 53, 57, 59, 61, 65, 67, 69, 71, 73, 75, 79, 81, 83,
85, 87, 89, 93, 95, 97, 101, 103, 107, 109, 111, 113, 115, 117,
 123, 125, 127, 129, 131, 135, 137, 139, 141, 145, 149, 151, 15
3, 155, 157, 159, 163, 167, 169, 171, 173, 177, 179, 181, 183,
185, 191, 193, 195, 197, 199, 201, 205, 207, 211, 213, 215, 219
, 221, 223, 225, 227, 229, 233, 235, 237, 239, 241, 243, 247, 2
49, 251, 255, 257, 261, 263, 265, 267, 269, 271, 277, 279, 281,
 283, 285, 289, 291, 293, 295, 299, 303, 305, 307, 309, 311, 31
3, 317, 321, 323, 325, 327, 331, 333, 335, 337, 339, 345, 347,
349, 351, 353, 355, 359, 361, 365, 367, 369, 373, 375, 377, 379
, 381, 383, 387, 389, 391, 393, 395, 397, 401, 403, 405, 409, 4
11, 415, 417, 419, 421, 423, 425, 431, 433, 435, 437, 439, 443,
 445, 447, 449, 453, 457, 459, 461, 463, 465, 467, 471, 475, 47
```



```
[+] p = 43 and q = 3
[+] n = 129 and euler totient = 84
[+] e = 55
[+] d = 55
[+] msg : 94
[+] Encrypted msg : 70
[+] Decrypted msg : 94
```



```
[+] p = 47 and q = 59
[+] n = 2773 and euler totient = 2668
[+] e = 2035
[1079]
[+] d = 1079
[+] msg : 680
[+] Encrypted msg : 1053
[+] Decrypted msg : 680
```

**OUTPUT EXPLANATION:**

The output shows the randomly generated prime numbers p and q, the calculated values of n and Euler's totient function (phi), the randomly generated values of e and d, the original message (msg), the encrypted message (e_msg), and the decrypted message (d_msg).

Since the values are generated randomly, the output will be different each time the code is run.

## Implementation: MD5

**CODE:**

```python
import hashlib



def main():

   msg = input("Enter message: ")

   hash = hashlib.md5()

   hash.update(msg.encode('utf-8'))


   print("MD5 Hash: {}".format(hash.hexdigest()))




if __name__ == "__main__":

   main()
```

**CODE EXPLANATION:**

The code imports the hashlib module, which provides various hash functions, including MD5.

The code defines a function named main(), which serves as the entry point of the program.

Inside the main() function, the user is prompted to enter a message using the input() function. The entered message is stored in the variable msg.

A new MD5 hash object is created using the hashlib.md5() function, and it is assigned to the variable hash.

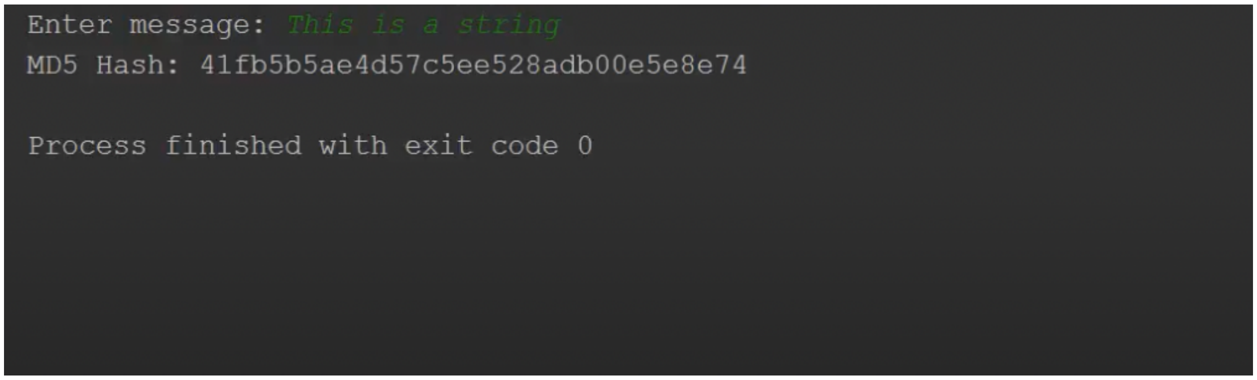
The update() method of the MD5 hash object is called to update the hash with the encoded version of the message. The encode() method is used to convert the string to bytes using the UTF-8 encoding.

Finally, the hexdigest() method is called on the MD5 hash object to obtain the hexadecimal representation of the hash. The result is printed to the console using the print() function.

The main() function is called at the end of the code using the if __name__ == "__main__": condition, which ensures that the main() function is only executed if the script is run directly and not imported as a module.

Overall, the code prompts the user to enter a message, calculates the MD5 hash of the message using the hashlib module, and prints the resulting hash value.

**OUTPUT:**



```
Enter message: This is a string
MD5 Hash: 41fb5b5ae4d57c5ee528adb00e5e8e74

Process finished with exit code 0
```

## Security Analysis: AES

Security Analysis of AES Implementation:

1. Potential Threats or Vulnerabilities:
    - Brute-Force Attack: An attacker may try all possible keys to decrypt the ciphertext. This can be mitigated by using a strong and sufficiently long key.
    - Side-Channel Attacks: Attackers may exploit information leaked through side channels like timing variations or power consumption to gain information about

    the key. Countermeasures like constant-time implementations or hardware protections can be used.
    - Known-Plaintext or Chosen-Plaintext Attacks: If an attacker has access to the plaintext and its corresponding ciphertext, they may attempt to derive the key. Care should be taken to prevent such information leakage.
2. Countermeasures and Best Practices:
    - Use a Strong Key: Choose a key that is long enough and generated from a cryptographically secure random source.
    - Protect Key Material: Safeguard the key used for encryption and decryption, ensuring it is stored securely and only accessible to authorized personnel.
    - Implement Encryption Correctly: Follow recommended implementation practices and standards for AES encryption to minimize vulnerabilities.
    - Secure Communication Channels: Ensure that the encrypted data is transmitted over secure channels to protect against interception or modification.
    - Regularly Update and Patch: Keep the implementation and supporting libraries up to date with the latest security patches to address any known vulnerabilities.
3. Limitations and Trade-offs:
    - Implementation Complexity: AES implementation can be complex, requiring careful attention to details to ensure security. Errors in the implementation can lead to vulnerabilities.
    - Performance Impact: Depending on the platform and implementation, AES encryption and decryption operations can be computationally intensive and may impact system performance.
    - Key Management: The security of the AES implementation relies heavily on proper key management practices, including key generation, storage, and distribution.

Conclusion:

Cryptography, including algorithms like AES, plays a crucial role in ensuring the confidentiality and integrity of sensitive information in cybersecurity and ethical hacking. However, it is essential to carefully implement and use cryptographic algorithms to mitigate potential vulnerabilities and threats. This requires a combination of strong encryption algorithms, secure key management practices, and continuous monitoring for emerging security risks. Regular security assessments and staying updated with the latest cryptographic best practices are crucial for maintaining the security of AES implementations and protecting sensitive data.

# Security Analysis: RSA

Security Analysis of RSA Implementation:

1. Potential Threats or Vulnerabilities:
    - Brute-Force Attack: An attacker may attempt to factorize the large prime numbers used in RSA to derive the private key. Countermeasures include using sufficiently large prime numbers.
    - Timing Attacks: Attackers may exploit timing variations during encryption or decryption to gather information about the key. Implementing constant-time operations can mitigate this threat.
    - Side-Channel Attacks: Leakage of information through power consumption or electromagnetic radiation can be exploited by attackers. Countermeasures like using secure hardware or implementing protections can address this vulnerability.
    - Key Management: Improper key storage or transmission can lead to key compromise and unauthorized access to encrypted data.
2. Countermeasures and Best Practices:
    - Key Length: Use long key lengths to make brute-force attacks computationally infeasible. Current recommendations for RSA key length are at least 2048 bits.
    - Random Number Generation: Use cryptographically secure random number generators to generate prime numbers and other cryptographic parameters.
    - Secure Key Storage: Protect the private key using strong encryption and access controls. Employ hardware security modules (HSMs) for enhanced key protection.
    - Secure Key Transmission: When sharing public keys, use secure channels to prevent interception or tampering.
    - Implement Padding Schemes: Utilize secure padding schemes (e.g., OAEP) to enhance the security of RSA encryption against attacks like chosen-ciphertext attacks.
3. Limitations and Trade-offs:
    - Performance Overhead: RSA operations can be computationally expensive, especially for large key sizes. Trade-offs may need to be made between security and performance requirements.
    - Key Management: Proper key management practices, including secure key generation, storage, and rotation, are crucial for maintaining the security of RSA implementations.

Conclusion:

RSA is a widely used asymmetric encryption algorithm that provides the foundation for secure communications and digital signatures. However, it is important to implement RSA correctly and follow best practices to mitigate potential vulnerabilities. This includes using long key lengths, employing secure random number generation, protecting private keys, and implementing secure key transmission. Additionally, addressing potential threats such as brute-force attacks, timing attacks, and side-channel attacks is crucial for ensuring the security of RSA implementations. Regular security audits, staying updated with cryptographic standards, and following recommended practices are essential for maintaining the integrity and confidentiality of RSA-based systems. Cryptography, including RSA, plays a critical role in cybersecurity and ethical hacking by providing secure communication channels, data protection, and authentication mechanisms.

## Security Analysis: MD5

Security Analysis of MD5 Implementation:

1. Potential Threats or Vulnerabilities:
    - Collision Attacks: MD5 is vulnerable to collision attacks, where different inputs can produce the same hash value. This can be exploited to forge documents, certificates, or malicious software.
    - Preimage Attacks: MD5 is susceptible to preimage attacks, where an attacker can find a message that generates a specific hash value. This can compromise the integrity of data.
    - Rainbow Table Attacks: Due to the deterministic nature of MD5, attackers can precompute and store hash values along with their corresponding inputs in a rainbow table. This allows for efficient reverse lookup and can compromise the security of hashed passwords.
2. Countermeasures and Best Practices:
    - Avoid MD5 Usage: It is recommended to avoid using MD5 for cryptographic purposes. Instead, adopt stronger hash functions like SHA-256 or SHA-3 for improved security.
    - Password Hashing: If MD5 must be used for password hashing, employ additional security measures like salting (adding a random value to each password before hashing) and multiple iterations (key stretching) to make the hashing process more resistant to attacks.
    - Upgrade Legacy Systems: If MD5 is currently used in legacy systems, consider upgrading to a more secure hash function to protect against modern attacks.
3. Limitations and Trade-offs:

- Performance: While MD5 is fast, its speed comes at the cost of security. Hash functions with stronger cryptographic properties may have a higher computational overhead.
- Cryptographic Weakness: MD5 was designed for checksum purposes, not for secure cryptographic operations. Its vulnerabilities make it unsuitable for applications requiring strong data integrity or security.

Conclusion:

MD5 is no longer considered secure for cryptographic purposes due to its vulnerabilities to collision attacks, preimage attacks, and rainbow table attacks. It is crucial to avoid using MD5 for any security-sensitive applications, especially for password storage or digital signatures. Instead, stronger hash functions like SHA-256 or SHA-3 should be employed. Upgrading legacy systems and adopting best practices, such as salting and key stretching, can provide an additional layer of security for MD5 usage. It is important to stay updated with current cryptographic standards and recommendations to ensure the integrity and confidentiality of data. Cryptography plays a vital role in cybersecurity and ethical hacking, and using strong and secure cryptographic algorithms is fundamental to maintaining data security and protecting against malicious attacks.