Sri Lanka Institute of Information Technology

Cryptographic Algorithm Evaluation

AES, RSA, SHA-256

Group Assignment – IE3081

IE3081_24_016

	Student ID	Student Name	Email	Contact Number
1	IT22357762	Dewmini P.L.T	IT22357762@my.sliit.lk	0741688488
2	IT22315496	Anuradha D.P.G.C	IT22315496@my.sliit.lk	0713952609
3	IT22002174	Dineth T.H.V.	IT22002174@my.sliit.lk	0766604096
4	IT22599186	S.Y.Jayasinghe	IT22599186@my.sliit.lk	0767096940

IE3081-Cryptograpghy

CONTRIBUTION

IT22357762	IT22315496	IT22002174	IT22599186
Documentation RSA – History Design principle Common user case Vulnerabilities	**Documentation** AES – History Design principle Common user case Vulnerabilities	**Documentation** SHA - History Design principle Common user case Vulnerabilities	**Documentation** AES – Performance characteristic RSA - Performance characteristic SHA - Performance characteristic
Implementation and Initial testing RSA implementation RSA, AES, SHA analyzing	**Implementation and Initial testing** AES implementation	**Implementation and Initial testing** SHA implementation	**Implementation and Initial testing** Perform encryption and decryption tests with sample data for all three algorithms and measure basic performance metrics.
Comprehensive testing RSA – Testing with different key sizes, input data sizes, and file types. Measure performance data (e.g., encryption time, CPU usage, memory usage).	**Comprehensive testing** AES – Testing with different key sizes, input data sizes, and file types. Measure performance data (e.g., encryption time, CPU usage, memory usage).	**Comprehensive testing** SHA – Testing with different key sizes, input data sizes, and file types. Measure performance data (e.g., encryption time, CPU usage, memory usage).	**Comprehensive testing** RSA , SHA , AES – Consolidate and visualize the performance data. Report down all the findings and compile the final report.

Contents

1. Introduction

Cryptographic algorithms are the fundamental to secure digital communication and data. As the technology is vastly moved with the ai and other latest technology, the threat landscape has increased. There by, the confidentiality, integrity and availability are protected converting the data into unreadable formats and verifying the origins. Encryption, decryption and the digital signatures are the techniques that the world of cryptography is used. In this report we provide a comprehensive idea on **AES** (symmetric key algorithm), **RSA** (asymmetric key algorithm), and **SHA-256** (hash function), three widely used cryptographic techniques. We aim to understand and evaluate the performance, security and efficiency under different conditions.

2. Overview and Evaluation

2.1 AES(Advanced Encryption Standard)

2.1.1 History

Advanced Encryption Standard (AES) is a symmetric block cipher developed by two Belgian cryptographers Joan Daemen and Vincent Rijmen , who submitted this encryption method during the NIST AES selection competition in 1997. That two cryptographers first they introduce this encryption method as a Rijndael. After the AES selection process this Rijndael method came to first place and NIST request from the that two Belgian cryptographers for rename their name to AES in 2001.

2.1.2 Design principal

In the AES architecture we mainly divide 2 parts as an encryption and decryption. In the encryption part plaintext convert to ciphertext. After this, the data suffer **Nr-1 rounds** of transformations. In that process First step is **AddRoundKey** where the plaintext XOR with a key. Next start the encryption round. In that round encryption part doing orderly **SubBytes** replace each bytes using substitution table , **ShiftRows** , **MixColumns** using linear transformation and **AddRoundKey** where the XOR again. In the last encryption round, the same operations occur, but **MixColumns** is forget, resulting in the ciphertext.

In the decryption process ciphertext convert to plaintext but reverse set of operations. Its start with **AddRoundKey** where the last round key is XORed with the ciphertext. Next **InvSubBytes** for reversing the byte substitution, **InvShiftRows** to reverse the row shifts. Finally start the decryption part and similar inversion step is used in the last decoding cycle, except that **InvMixColumns** is skipped. After these rounds, the ciphertext is decrypted and the original plaintext is recovered.

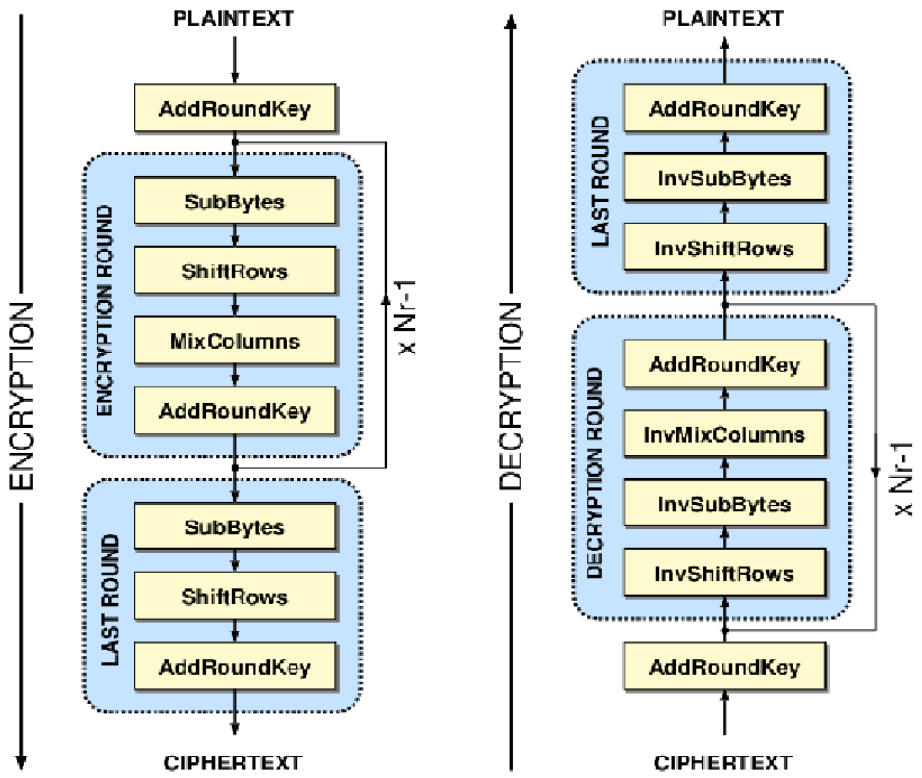

Figure 01 - The basic AES-128 cryptographic architecture

2.1.3 Common use case

Data encryption - Used to encrypt communications data, databases and files.

TLS/SSL - AES is frequently used to secure web connections using the SSL/TLS protocols.

VPNs -Virtual private networks use AES to protect data transfer.

Disk encryption - BitLocker and File Vault, two full disk encryption solutions, use AES.

2.1.4 Vulnerabilities

Side Channel Attacks -Like other encryption algorithms, AES is vulnerable to side-channel attacks that use implementation-specific information (such as power usage or electromagnetic leakage).

Key management vulnerability - Inefficient key management can compromise AES encryption. Unauthorized access can be caused by weak or exposed keys.

Error Attacks(fault injection attack) - These include deliberately introducing flaws into the encryption process so that attackers can recover the encryption keys.

The Quantum Threat - AES is still safe from conventional computers but may be a concern for quantum computers in the future. Nevertheless, AES is believed to offer some resilience with larger key sizes (like AES-256).

2.1.5 Performance characteristics

Attribute	Description
AES Speed	Fast encryption and decryption.
Efficiency in Hardware	Highly efficient in hardware, with built-in support in many processors, allowing rapid encryption/decryption with minimal performance overhead.
Efficiency in Software	Efficient in software implementations, especially with AES-NI hardware acceleration, making it suitable for a wide range of platforms, from servers to mobile devices.
Memory Requirements	Relatively low memory requirements, suitable for embedded systems of all sizes.
Block Size	key sizes of 128, 192, and 256 bits

2.2 RSA Algorithm

2.3 History

This system was invented by Ron Rivest, Adi Shamir, and Len Adleman in 1977. RSA cryptography is based on a public-key-based cryptosystem. it is the most widely used cryptography algorithm in the world. We can use it to encrypt a message without needing a secret key exchange. This algorithm can be used for both public key encryption and digital signatures. The security of RSA is based on the difficulty of solving certain mathematical problems such as factoring large integers.

2.3.1 Design principal

RSA involve with the key principles of,

01. **Prime Factorization**

The security of RSA's relies on the challenge of factoring the product of two large prime numbers (p and q) used to generate the keys.

02. **Public Key and Private Key**

The public key can be used to encrypt the messages while only the private key used to decrypt the messages.

03. **Mathematical One-Way Function**

Because multiplying two prime numbers is simple, factoring the product is computationally impossible for big numbers. This is the basis for RSA's use of modular exponentiation.

04. **Key Generation**

Creating a pair of cryptographic keys, a public key for encryption and signature verification, and a private key for decryption and signing. This process is based on the difficulty of factoring large numbers. By doing this it is computationally secure under current technology.

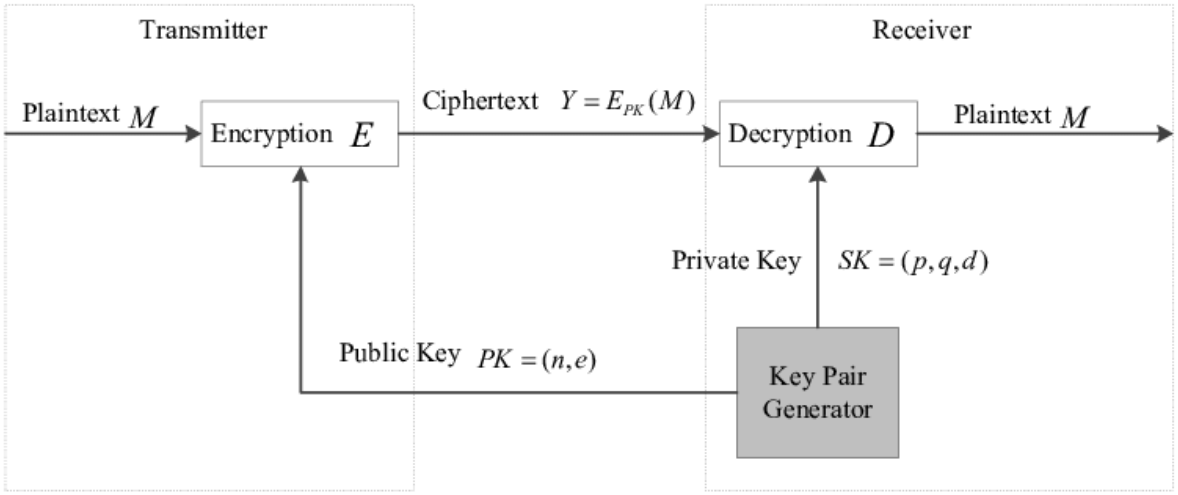

Figure 02 - The basic RSA cryptographic architecture

2.3.2 Common use case

Data Encryption -Sensitive data is frequently encrypted using RSA. RSA is frequently used to encrypt small data blocks, similar to symmetric keys in hybrid encryption systems, despite its inefficiency for large-scale data encryption.

Digital Signatures-To ensure data integrity, authenticity, and non-repudiation RSA can used. A sender can sign a document using their private key and the recipient can confirm it with the sender's public key.

Secure Key Exchange- SSL/TSL is commonly used to exchange keys securely between parties. It guarantees the safe sharing of symmetric encryption keys during encrypted sessions.

Authentication - Verifying identities and guaranteeing the integrity of data transferred are two uses for RSA, such as in SSH key-based authentication.

2.3.3 Vulnerabilities

Key Size Vulnerability- A key that is too tiny (512 bits, for example) could expose RSA to brute-force attacks. Key sizes in modern systems are typically 2048 bits or larger.

Chosen-Ciphertext Attack-To decipher or alter messages, attackers can transmit specially constructed ciphertexts. Padding techniques like OAEP (Optimal Asymmetric Encryption Padding) are frequently used to prevent this.

Timing Attack-Variations in the amount of time needed to complete decryption operations can be used by side-channel attackers to obtain the private key. Timing attack mitigations entail making sure that processes take an even amount of time.

Quantum Computing Threat (Factoring Vulnerability)-The main component contributing to RSA's security is how hard it is to factor huge integers. Technological developments in quantum computing (like as Shor's algorithm) have the potential to break RSA by effectively factoring big numbers.

2.3.4 Performance characteristics

Performance Characteristic	Description
Encryption and Decryption Speed	RSA encryption with the public key is fast (due to small exponents like 65537), but decryption with the private key is computationally expensive. To optimize performance, RSA is often combined with symmetric key cryptography, where RSA encrypts the symmetric key, and the symmetric key encrypts the data.
Key Size Impact	RSA slows down as the key size increases. While 2048-bit keys are recommended for general use, 3072-bit or 4096-bit keys provide better long-term security but result in slower operations.
Memory and Computational Requirements	Large key sizes in RSA demand significant memory and processing power due to the intensity of key generation, encryption, and decryption processes.
Quantum Resistance	RSA is vulnerable to attacks by quantum computers, as algorithms like Shor’s could break RSA by making the factorization of large integers trivial.

2.4 SHA-256 (Secure Hash Algorithm)

Secure Hash Algorithm(SHA-256) is a cryptographic function that transform the data into hash functions. Hash function is a fixed length string which is one -way making impossible to revert it back. This technique is widely used for store password in server side.

2.4.1 History

SHA – 256 is a part of SHA-2 family which was developed by **National Security Agency(NSA)** and introduced by the **National Institute of Standards and Technology (NIST)** as a part of the **Digital Signature Standard (DSS).** This was improved on SHA-1, released in 1995 as it had a vulnerability in collisions where make the same has value for two different inputs. This was introduced in 2001 with different output sizes including 224, 256, 384, and 512 bits. Among them, SHA 256 became the most widely adopted because it's security level and high computational power. Regardless of the input size it output 256 bits hash value.

2.4.2 Design principal

This hash function is designed to create 256 bits hash value which is a fixed length. Regardless of the key size, data divided into blocks and iteratively processes each to output the hash value with fixed length. As the initial process, the data has to go through 64 rounds of iterations of iterations to enhance the complexity. In each round data is shifted by a number of positions. Nevertheless, addictive constants which is added to the data each round making it random and unpredicted. At the same time, bitwise logical operators including XOR, OR, and AND are applied. With that process, even a tiny change in the input make completely different hash values. Each round adds extra level of security and implement collision resistance.

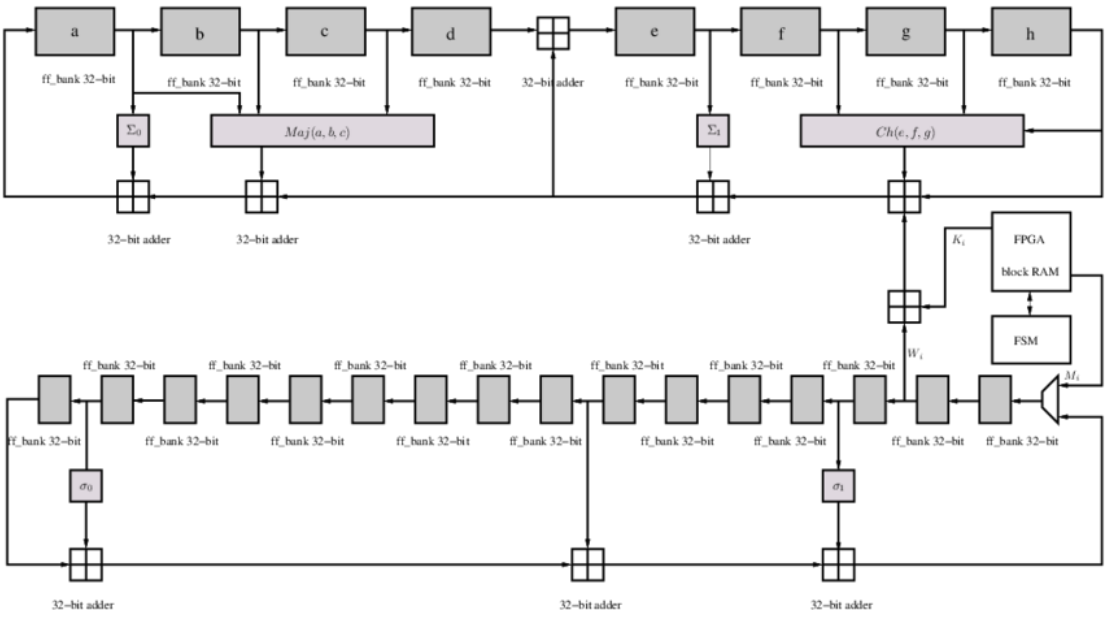

Figure 03 – The implementation of SHA-256

2.4.3 Common use case

SHA 256 is in use of different contexts due to its strong security properties.

Data Integrity - This has widely used in file integrity checks, ensuring files have not altered. Tiny modification outputs completely different output as earlier mentioned. Because of that, comparing both hashes of the original and received data allows to check the integrity of the data.

Cryptocurrency – Mostly bitcoin on other cryptocurrencies rely on SHA-256 for mining and securing transactions within blockchain technology.

Digital Signatures - To verify the authenticity and integrity of digital documents, SHA -256 is used. It creates a hash value of the content to be signed and encrypt that with a private key. With the use of decryption key which is a public key decrypt it back ensuring the integrity and identity.

Secure Communications – For the purpose of having secure communication, SHA 256 is embedded in TLS and SSL protocols.

Password Hashing - When in storing the password in server side, SHA 256 and salt value is added to convert it into secure hash values.

2.4.4 Vulnerabilities

With the root search, any known security vulnerabilities or computational attacks were found. But there exit some theoretical attacks of collision finding two inputs with same hash value. But it has limited to the theory only as it requires 2^128 computational power. Nevertheless, pre-imaged attacks where involve in finding an input for specific hash are also limited to books as SHA 256 has large output space. But when considering the recent research on quantum computing, there might be a possibility in breaking the collision resistance. For that security problem, quantum-safe cryptographic methods are in the researching level.

2.4.5 Performance characteristics

Performance Characteristic	Description
Hash Output Size	256 bits (32 bytes)
Speed	Relatively fast for smaller inputs but slower than SHA-1
Collision Resistance	Highly resistant and no known real-world collisions
Pre-image Resistance	Strong; computationally infeasible to reverse the hash
Computational Overhead	Moderate CPU and memory usage
Efficiency	Balanced security and speed for various applications
Security	Can be targeted of quantum computing attacks in future

3. Implementation and Initial Testing

3.1 AES Implementation

```python
def decrypt_in_parallel(ciphertext_chunks, key, iv):                                    #Decryption of multiple chunks
    with ThreadPoolExecutor() as executor:
        decrypted_chunks = list(executor.map(aes_decrypt_chunk, ciphertext_chunks, [key] * len(ciphertext_chunks), [iv] * len(ciphertext_chunks)))
    return b''.join(decrypted_chunks)


def chunk_data(data, chunk_size=1024 * 4):                                              #Function which split data into chunks
    return [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]

#.....................................................Main function.....................................................................................
def main():

    x = input("Enter a message to encrypt: ")                                   #Input the plain text


    p_text = x.encode('utf-8')                                                  #Convert user input to bytes


    p_text_chunks = chunk_data(p_text)                                          #Split into chunks


    key = get_random_bytes(16)                                                  #Random key generation


    start_time = time.perf_counter()                                            #Encryption time
    encrypted_data = encrypt_in_parallel(p_text_chunks, key)
    encryption_time = time.perf_counter() - start_time
    print(f"Parallel Encryption time: {encryption_time:.6f} seconds")


    iv = encrypted_data[:16]                                                   #Extraction of IV from encrypted data
    c_text = encrypted_data[16:]
```

```
C: > Users > USER > Downloads > aes.py
1   from Crypto.Cipher import AES                                               #Import AES library
2   from Crypto.Random import get_random_bytes                                  #Import the function to generate the random bytes
3   from concurrent.futures import ThreadPoolExecutor                           #Import the funtion to parallel processing
4   import time
5
6
7   def aes_encrypt_chunk(chunk, key, iv):                                      #The function of chunking data into 16 bytes blocks
8       cipher = AES.new(key, AES.MODE_CBC, iv)
9       return cipher.encrypt(pad(chunk))                                       #The padded function is used ensure the data is with correct length.
10
11
12  def aes_decrypt_chunk(ciphertext, key, iv):                                 #AES decryption function implementation
13      cipher = AES.new(key, AES.MODE_CBC, iv)
14      return unpad(cipher.decrypt(ciphertext))
15
16
17  def pad(plaintext):                                                         #Padding function to ensure each chunk set is with 16 bytes.
18      padding_length = 16 - len(plaintext) % 16
19      return plaintext + bytes([padding_length] * padding_length)
20
21
22  def unpad(plaintext):                                                       #Function to remove the padding when decrypting
23      padding_length = plaintext[-1]
24      return plaintext[:-padding_length]
25
26
27  def encrypt_in_parallel(plaintext_chunks, key):                             #Parallel Encryption
28      iv = get_random_bytes(16)                                               #Generate IV once.
29      with ThreadPoolExecutor() as executor:
30          encrypted_chunks = list(executor.map(aes_encrypt_chunk, plaintext_chunks, [key] * len(plaintext_chunks), [iv] * len(plaintext_chunks)))
31      return iv + b''.join(encrypted_chunks)
32
33
34  def decrypt_in_parallel(ciphertext_chunks, key, iv):                        #Decryption of multiple chunks
```

```python
def main():


    c_text_chunks = chunk_data(c_text)                                          #Ciphertext into chunks

    start_time = time.perf_counter()                                            #Time calculation for decryption process
    decrypted_data = decrypt_in_parallel(c_text_chunks, key, iv)
    decryption_time = time.perf_counter() - start_time
    print(f"Parallel Decryption time: {decryption_time:.6f} seconds")


    print(f"Decrypted message: {decrypted_data.decode('utf-8')}")           #Display the decrypted message which is identical to the entered message.

if __name__ == "__main__":
    main()
```

3.1.1 Output

```
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: a
Parallel Encryption time: 0.001447 seconds
Parallel Decryption time: 0.000704 seconds
Decrypted message: a
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: ab
Parallel Encryption time: 0.001399 seconds
Parallel Decryption time: 0.000543 seconds
Decrypted message: ab
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abc
Parallel Encryption time: 0.001953 seconds
Parallel Decryption time: 0.001308 seconds
Decrypted message: abc
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abcd
Parallel Encryption time: 0.001646 seconds
Parallel Decryption time: 0.000993 seconds
Decrypted message: abcd
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abcde
Parallel Encryption time: 0.001359 seconds
Parallel Decryption time: 0.000680 seconds
Decrypted message: abcde
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abcdef
Parallel Encryption time: 0.001191 seconds
Parallel Decryption time: 0.000821 seconds
Decrypted message: abcdef
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abscdhkekjfrhwgkerlwfijeofhia
Parallel Encryption time: 0.001596 seconds
Parallel Decryption time: 0.001331 seconds
Decrypted message: abscdhkekjfrhwgkerlwfijeofhia
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: hjeGLVWUAGLAUIHQIUGAEIWQGeiGLVHLWHRLIHVuirgfegfviuhelihdiohdoqhrfUIQGFIUQEDwhjgfygqE
Parallel Encryption time: 0.001454 seconds
Parallel Decryption time: 0.000864 seconds
Decrypted message: hjeGLVWUAGLAUIHQIUGAEIWQGeiGLVHLWHRLIHVuirgfegfviuhelihdiohdoqhrfUIQGFIUQEDwhjgfygqE
```

This outputs pointer out how the encryption and decryption time differ from each with the respective input size.

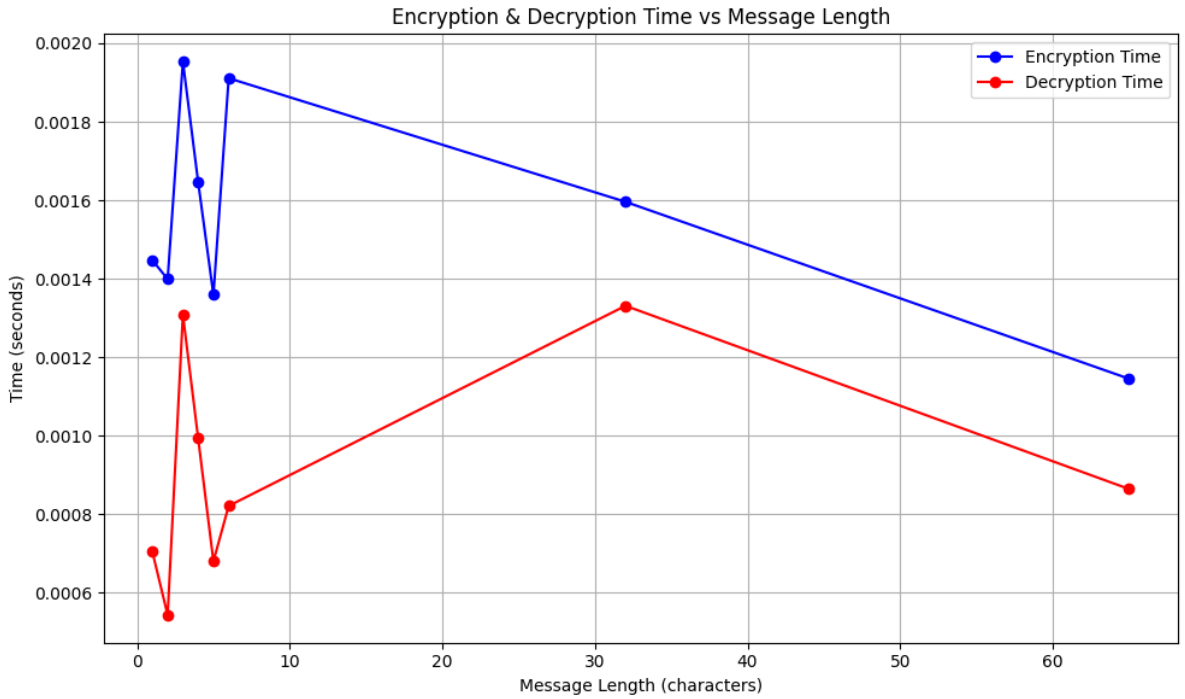

Figure 04 :AES Encryption and Decryption Time VS Message Length

For very short messages (0-5 characters) –

- Time spent encrypting: The variation is substantial. For a message with three to four characters, the encryption time varies, peaking at 0.002 seconds.
  Time needed to decrypt: There are many peaks and troughs in the decryption time, which first peaks at about 0.0012 seconds at about 4 characters before going down rapidly and then increasing again.

As message length increases from 5 to 10 characters –

- Time spent encrypting: The general trend starts to level out, but there is another oscillation.
  Time needed to decrypt: There is a minor inefficiency at this point, since the decryption time varies once again and peaks again at 10 characters, when it increases to around 0.0015 seconds.

For messages between 10 to 30 characters-

- Time spent encrypting: As message length rises, the pattern smoothed out and gradually shows a decrease in encryption time.
  Time needed to decrypt: After ten characters, there is a tiny increase, suggesting that the decryption time increases slightly with message length.

For messages between 30 and 60 characters:

- Time spent encrypting: There is a noticeable declining tendency. As the message length grows, the encryption time steadily decreases. As the message length increases above this range, this suggests a more effective encryption procedure.
  Time needed to decrypt: Beyond 50 characters, the decryption time similarly shows a decreasing trend, notwithstanding a modest increase after 30 characters.

Efficiency observation:

- As the message length rises over 30–40 characters, both encryption and decryption appear to function better, indicating that the algorithm manages bigger data more effectively.

```
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abcdefghijklmnop
Parallel Encryption time: 0.001666 seconds
Parallel Decryption time: 0.001076 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abcdefghijklmnop
Parallel Encryption time: 0.001485 seconds
Parallel Decryption time: 0.000936 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abcdefghijklmnop
Parallel Encryption time: 0.001309 seconds
Parallel Decryption time: 0.000961 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abcdefghijklmnop
Parallel Encryption time: 0.001635 seconds
Parallel Decryption time: 0.000806 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads> python3.8 aes.py
Enter a message to encrypt: abcdefghijklmnop
Parallel Encryption time: 0.001492 seconds
Parallel Decryption time: 0.000908 seconds
Decrypted message: abcdefghijklmnop
```

This shows how the time differ even giving the same input.

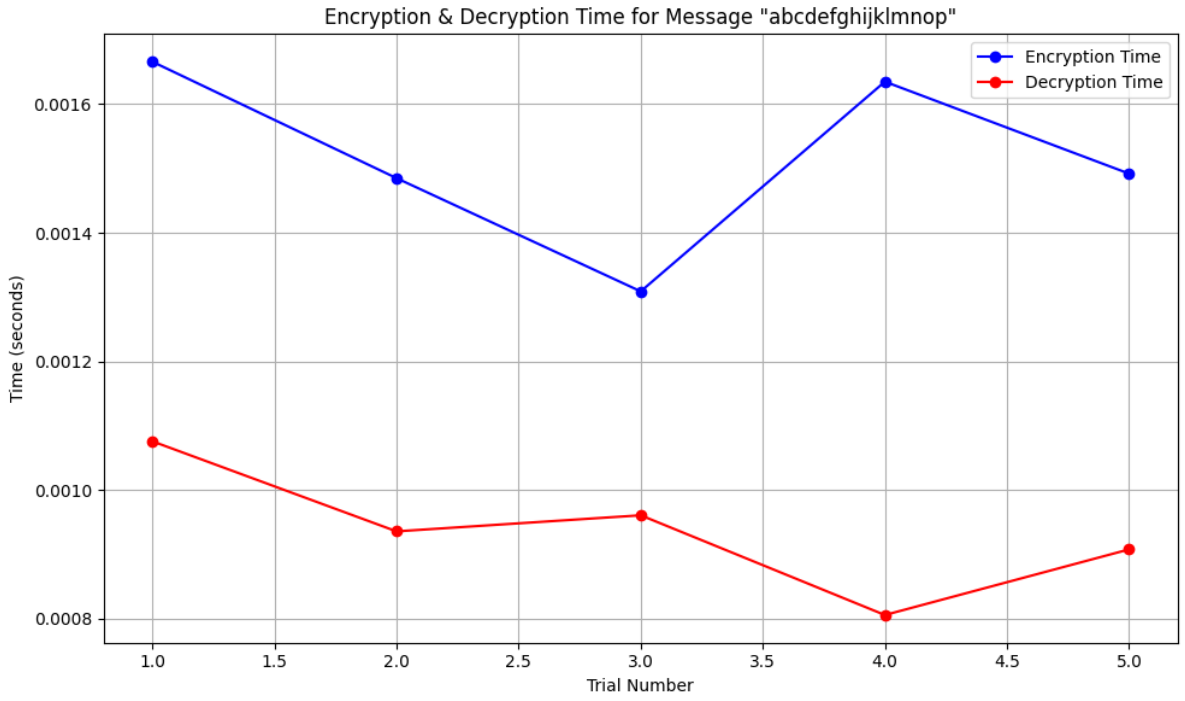

Figure 05 :AES Encryption and Decryption Time VS Message 'abcdefghijklmnop'

First Trial:

Time spent encrypting: At about 0.0016 seconds, the encryption time is quite high at first.
Time needed to decrypt: At about 0.0010 seconds, the decryption time is less than the encryption time.

Second Trial:

Time spent encrypting: The encryption time gradually decreases, reaching around 0.0013 seconds.
Time needed to decrypt: the decryption time decreases little, reaching 0.0009 seconds.

Third Trial:

Time spent encrypting: At around 0.0012 seconds, the encryption time reaches its lowest point after continuing to decline. Out of all the trials, this one seems to have the fastest encryption time.
Time needed to decrypt: The decryption time has increased little, surpassing 0.0009 seconds.

Trial four:

Time spent encrypting: After the prior decrease, the encryption time peaks at about 0.0015 seconds. It then climbs once again.
Time needed to decrypt: During this attempt, the decryption time approaches 0.0008 seconds, which is the lowest figure compared to encryption.

Trial five:

Time spent encrypting: Once more, the encryption duration drops to about 0.0014 seconds, suggesting a little higher efficiency than in Trial 4.
Time needed to decrypt: Despite a little increase to around 0.0009 seconds, the decryption time is still rather short.

3.2 RSA Implementation

```python
from Crypto.PublicKey import RSA                                                    # Import RSA library
from Crypto.Cipher import PKCS1_OAEP                                                # Import the function for RSA encryption/decryption
from concurrent.futures import ThreadPoolExecutor                                   # Import the function for parallel processing
import time                                                                         # Import time measuring function


def generate_rsa_keys():
    key = RSA.generate(2048)                                                        # Key generation
    pr_key = key
    pu_key = key.publickey()
    return pr_key, pu_key


def rsa_encrypt(pl_text, pu_key):                                                   # RSA encryption function
    cipher = PKCS1_OAEP.new(pu_key)
    return cipher.encrypt(pl_text)


def rsa_decrypt(ciphertext, pr_key):                                                # RSA decryption function
    cipher = PKCS1_OAEP.new(pr_key)
    return cipher.decrypt(ciphertext)


def parallel_rsa_encrypt(plaintext_chunks, pu_key):                                 # Function for parallel encryption of multiple chunks
    with ThreadPoolExecutor() as executor:
        encrypted_chunks = list(executor.map(rsa_encrypt, plaintext_chunks, [pu_key] * len(plaintext_chunks)))
    return b''.join(encrypted_chunks)


def parallel_rsa_decrypt(ciphertext_chunks, pr_key):                                # Function for parallel decryption of multiple chunks
    with ThreadPoolExecutor() as executor:
        decrypted_chunks = list(executor.map(rsa_decrypt, ciphertext_chunks, [pr_key] * len(ciphertext_chunks)))
    return b''.join(decrypted_chunks)
```

```python
def chunk_data(data, chunk_size=245):                                               # Function to split the data into chunks
    return [data[i:i + chunk_size] for i in range(0, len(data), chunk_size)]


def main():
    user_input = input("Enter a message to encrypt: ")                              # User input

    plaintext = user_input.encode('utf-8')                                          # Convert the input message to bytes (UTF-8)

    plaintext_chunks = chunk_data(plaintext)                                        # Split the plaintext into 245-byte chunks

    pr_key, pu_key = generate_rsa_keys()                                            # Generate RSA public and private keys

    start_time = time.perf_counter()                                                # Start time measurement for encryption
    ciphertext = parallel_rsa_encrypt(plaintext_chunks, pu_key)                     # Perform parallel RSA encryption
    encryption_time = time.perf_counter() - start_time                              # Calculate the encryption time
    print(f"RSA Parallel Encryption time: {encryption_time:.6f} seconds")           # Print the time taken for encryption

    ciphertext_chunks = chunk_data(ciphertext, 256)                                 # Split the ciphertext into 256-byte chunks

    start_time = time.perf_counter()                                                # Start time measurement for decryption
    decrypted_message = parallel_rsa_decrypt(ciphertext_chunks, pr_key)             # Perform parallel RSA decryption
    decryption_time = time.perf_counter() - start_time                              # Calculate the decryption time
    print(f"RSA Parallel Decryption time: {decryption_time:.6f} seconds")           # Print the time taken for decryption

    print(f"Decrypted message: {decrypted_message.decode('utf-8')}")                # Display the decrypted message


# Entry point for the script
if __name__ == "__main__":
    main()
```

```
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: a
RSA Parallel Encryption time: 0.001184 seconds
RSA Parallel Decryption time: 0.004434 seconds
Decrypted message: a
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: ab
RSA Parallel Encryption time: 0.001699 seconds
RSA Parallel Decryption time: 0.004322 seconds
Decrypted message: ab
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abc
RSA Parallel Encryption time: 0.002337 seconds
RSA Parallel Decryption time: 0.008088 seconds
Decrypted message: abc
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abcd
RSA Parallel Encryption time: 0.001510 seconds
RSA Parallel Decryption time: 0.004270 seconds
Decrypted message: abcd
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abcde
RSA Parallel Encryption time: 0.001421 seconds
RSA Parallel Decryption time: 0.004153 seconds
Decrypted message: abcde
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abcdef
RSA Parallel Encryption time: 0.001423 seconds
RSA Parallel Decryption time: 0.004388 seconds
Decrypted message: abcdef
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abscdhkekjfrhwgkerlwfijeofhia
RSA Parallel Encryption time: 0.002887 seconds
RSA Parallel Decryption time: 0.009777 seconds
Decrypted message: abscdhkekjfrhwgkerlwfijeofhia
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: hjeGLVWUAGLAUIHQIUGAEIWQGeiGLVHLWHRLIHVuirgfegfviuhelihdiohdoqhrfUIQGFIUQEDwhjgfygqE
RSA Parallel Encryption time: 0.001433 seconds
RSA Parallel Decryption time: 0.004767 seconds
Decrypted message: hjeGLVWUAGLAUIHQIUGAEIWQGeiGLVHLWHRLIHVuirgfegfviuhelihdiohdoqhrfUIQGFIUQEDwhjgfygqE
PS C:\Users\USER\Downloads>
```

This outputs pointer out how the encryption and decryption time differ from each with the respective input size.

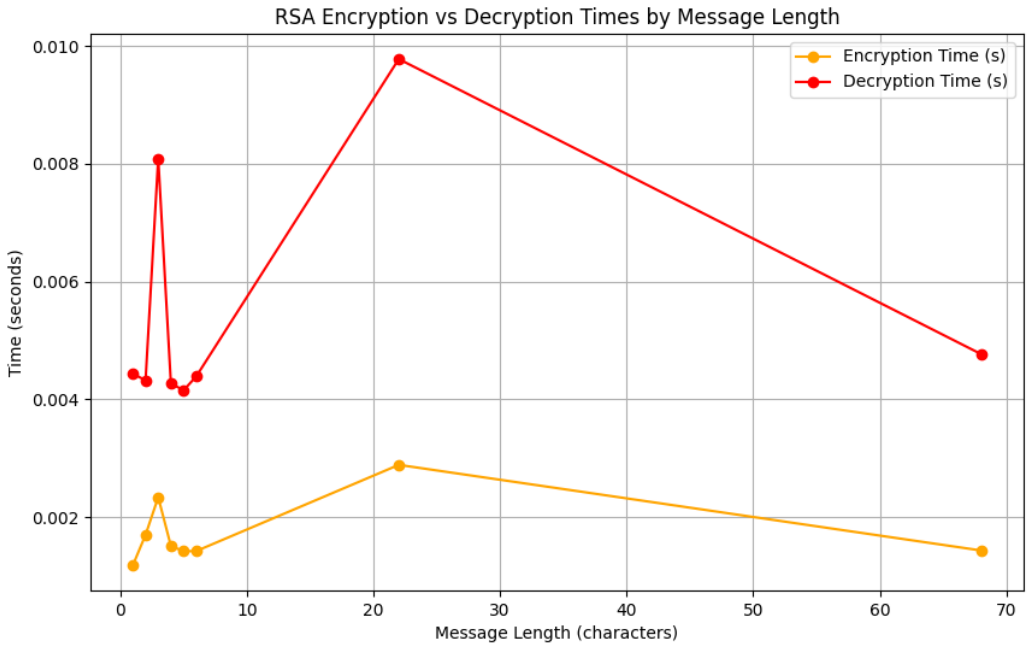

Figure 06 :RSA Encryption and Decryption Time VS Message Length

Time spent encrypting:

stays low over time, showing that RSA works well for encrypting communications of different lengths with very slight variations.

Time needed to decrypt:

show considerable variation, particularly for shorter message lengths, peaking at about 20 characters, at which point the decryption time increases noticeably. However, as the message length surpasses 20 characters, the decryption procedure becomes more effective.

While decryption durations are longer for short to mid-length messages but become better with bigger message sizes, RSA exhibits consistent encryption speeds over a range of message lengths.

```
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abcdefghijklmnop
RSA Parallel Encryption time: 0.001876 seconds
RSA Parallel Decryption time: 0.004466 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abcdefghijklmnop
RSA Parallel Encryption time: 0.001614 seconds
RSA Parallel Decryption time: 0.005586 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abcdefghijklmnop
RSA Parallel Encryption time: 0.001639 seconds
RSA Parallel Decryption time: 0.004045 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abcdefghijklmnop
RSA Parallel Encryption time: 0.001495 seconds
RSA Parallel Decryption time: 0.004291 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads> python3.8 RSA.py
Enter a message to encrypt: abcdefghijklmnop
RSA Parallel Encryption time: 0.001713 seconds
RSA Parallel Decryption time: 0.004383 seconds
Decrypted message: abcdefghijklmnop
PS C:\Users\USER\Downloads>
```

This shows how the time differ even giving the same input.

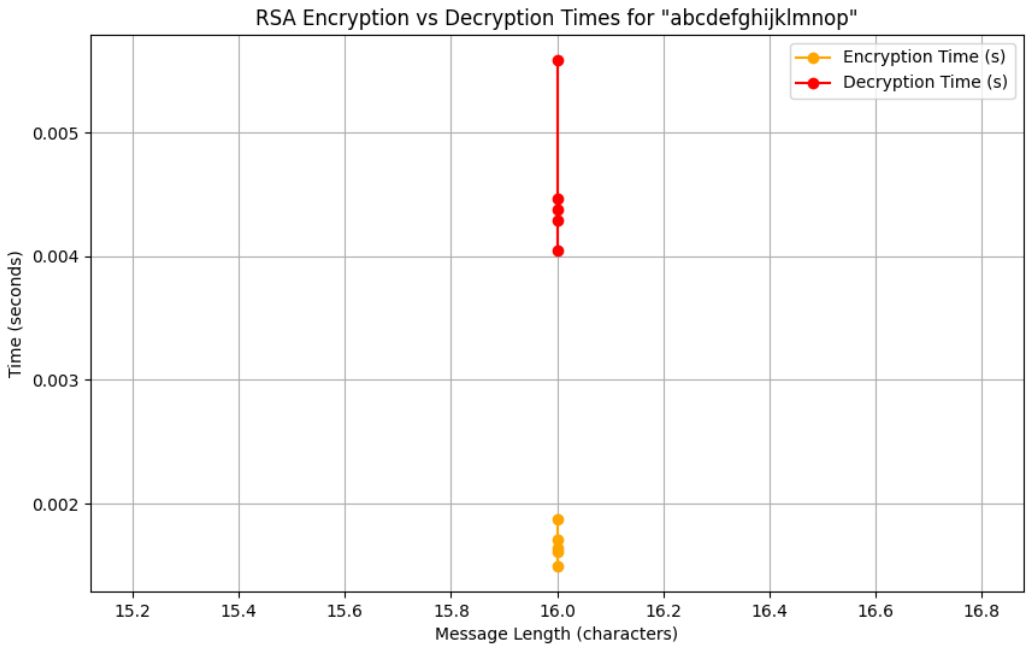

Figure 07 :RSA Encryption and Decryption Time VS Message 'abcdefghijklmnop'

With highly small deviations, the message length is set at around 16 characters, as seen on the x-axis.

RSA encryption is quick and reliable for messages up to 16 characters long, but decryption takes a lot longer. Although both encryption and decryption exhibit a consistent pattern across several tests, the decryption procedure proceeds more slowly, which is to be anticipated given the computational cost of RSA.

3.3 SHA – 256 Implementation

```python
import hashlib                                                          #Import hash library
from concurrent.futures import ThreadPoolExecutor                       #Import the function for parallel processing
import time                                                             #Import the function time

                                                                        #Hashing function for single chunk
def sha256_hash_chunk(data_chunk):
    sha256 = hashlib.sha256()
    sha256.update(data_chunk)
    return sha256.digest()


                                                                        #Parallel hashing function
def sha256_parallel(data, chunk_size=1024*1024):
    chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]            # Split the data into 1 MB chunks


    with ThreadPoolExecutor() as executor:
        results = list(executor.map(sha256_hash_chunk, chunks))


                                                                        # Combine all results into one final SHA-256
    final_sha256 = hashlib.sha256()
    for chunk_digest in results:
        final_sha256.update(chunk_digest)

    return final_sha256.hexdigest()


def main():

    message = input("Enter the message to hash: ")                      # User Input


    data = message.encode('utf-8')                                      # Convert the input message to bytes (UTF-8)
```

```python
                                                                        # Measure the time for final hashing
start_time = time.perf_counter()
hash_value = sha256_parallel(data)
hash_time = time.perf_counter() - start_time

print(f"SHA-256 Hash: {hash_value}")
print(f"Hashing time: {hash_time:.6f} seconds")


f __name__ == "__main__":
    main()
```

```
Enter the message to hash: a
SHA-256 Hash: bf5d3affb73efd2ec6c36ad3112dd933efed63c4e1cbffcfa88e2759c144f2d8
Hashing time: 0.001026 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: ab
SHA-256 Hash: a1ff8f1856b5e24e32e3882edd4a021f48f28a8b21854b77fdef25a97601aace
Hashing time: 0.000939 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abc
SHA-256 Hash: 4f8b42c22dd3729b519ba6f68d2da7cc5b2d606d05daed5ad5128cc03e6c6358
Hashing time: 0.000603 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abcd
SHA-256 Hash: 7e9c158ecd919fa439a7a214c9fc58b85c3177fb1613bdae41ee695060e11bc6
Hashing time: 0.001020 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abcde
SHA-256 Hash: 1d72b6eb7ba8b9709c790b33b40d8c46211958e13cf85dbcda0ed201a99f2fb9
Hashing time: 0.001004 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abcdef
SHA-256 Hash: ce65d4756128f0035cba4d8d7fae4e9fa93cf7fdf12c0f83ee4a0e84064bef8a
Hashing time: 0.001046 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abscdhkekjfrhwgkerlwfijeofhia
SHA-256 Hash: b0014a5284d950068285360dfde10c3e0ec55d830490886968a54733178d7c29
Hashing time: 0.001011 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: hjeGLVWUAGLAUIHQIUGAEIUGAEIWQGeiGLVHLWHRLIHVuirgf
egfviuhelihdiohdoqhrfUIQEDwhjgfygqE
SHA-256 Hash: 533fba1a957424176128a49ec8d74c3e9d82b24949db18014795f60ec7b7038c
Hashing time: 0.001007 seconds
PS C:\Users\USER\downloads>
```

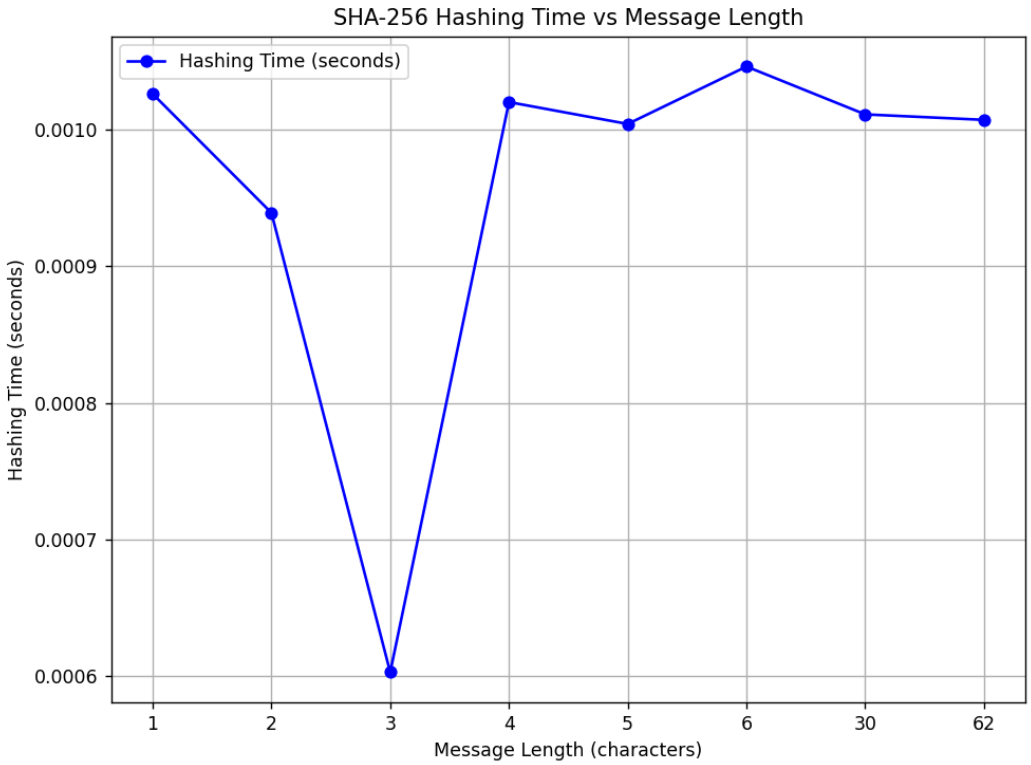

Figure 08 :SHA-256 Hashing Time VS Message Length

First variations:

For shorter messages (1-6 characters):

The hashing time varies greatly; the shortest time is at 3 characters. This drop might be the result of optimizations or system-specific variables that make hashing especially effective at that particular length.

Brief messages (one to six characters):

Display considerable variation in hashing time, with a discernible decrease at three characters.

Messages with more characters (30+):

For larger messages, hashing times level out and the procedure becomes reliable.

Longer messages are handled effectively by the SHA-256 algorithm, which sees no change in hashing time as message length grows. It is possible that system-specific characteristics are the cause of the exceptional efficiency shown for very brief communications.

```
Enter the message to hash: abcdefghijklomnop
SHA-256 Hash: d5439526e2b3088ae7442914bdb4d830c75be29e92849f57085d319fc60af7e3
Hashing time: 0.001053 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abcdefghijklomnop
SHA-256 Hash: d5439526e2b3088ae7442914bdb4d830c75be29e92849f57085d319fc60af7e3
Hashing time: 0.001143 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abcdefghijklomnop
SHA-256 Hash: d5439526e2b3088ae7442914bdb4d830c75be29e92849f57085d319fc60af7e3
Hashing time: 0.000888 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abcdefghijklomnop
SHA-256 Hash: d5439526e2b3088ae7442914bdb4d830c75be29e92849f57085d319fc60af7e3
Hashing time: 0.001156 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abcdefghijklomnop
SHA-256 Hash: d5439526e2b3088ae7442914bdb4d830c75be29e92849f57085d319fc60af7e3
Hashing time: 0.000938 seconds
PS C:\Users\USER\downloads> python3.8 SHA.py
Enter the message to hash: abcdefghijklomnop
SHA-256 Hash: d5439526e2b3088ae7442914bdb4d830c75be29e92849f57085d319fc60af7e3
Hashing time: 0.001047 seconds
PS C:\Users\USER\downloads>
```

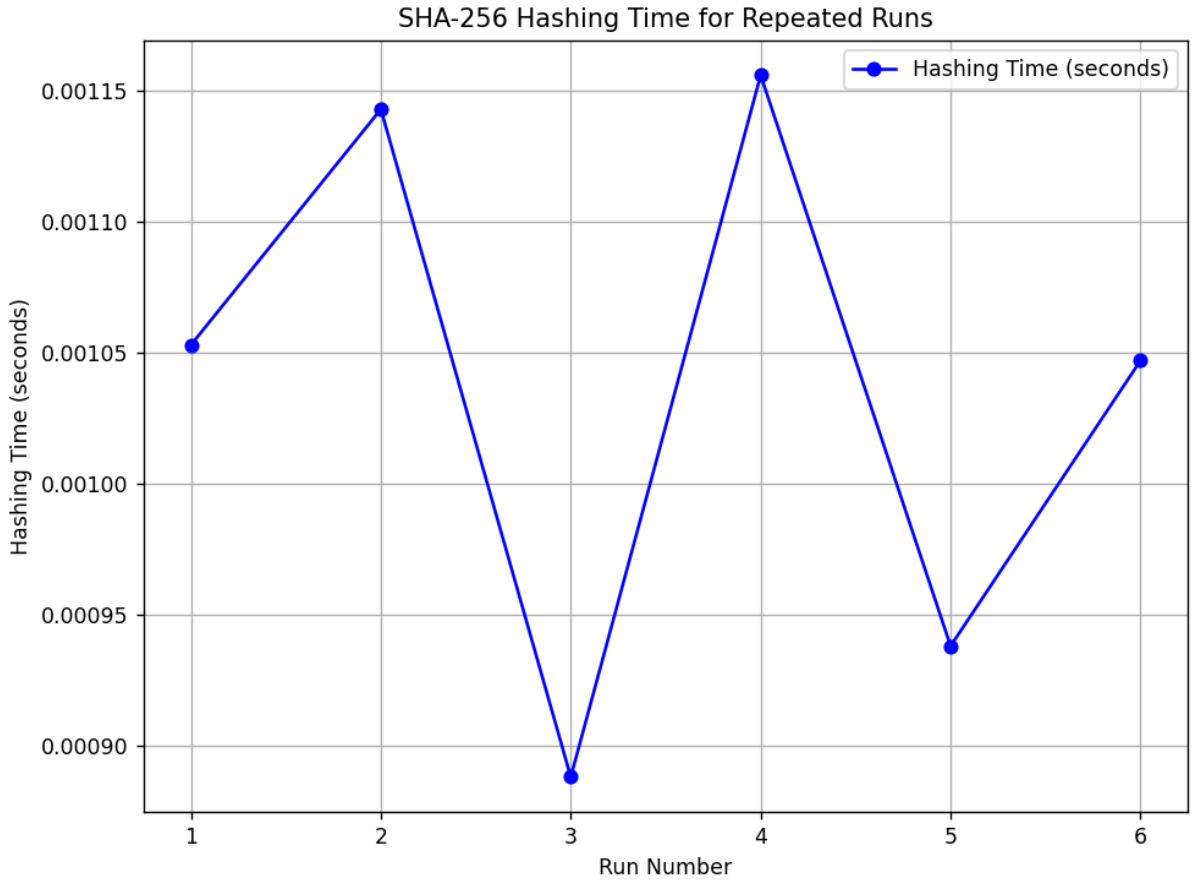

Figure 09 :SHA-256 Hashing Time VS Message 'abcdefghijklmnop'

There are noticeable peaks and troughs in the SHA-256 hashing process, which exhibits some unpredictability between repeated runs. The variations might be explained by the state of the system at the time of testing. Notwithstanding these variances, the hashing time as a whole

remains within a narrow range, showing steady performance over time for repeated hashing operations.

4. Comprehensive Testing and Performance Analysis

4.1 AES

4.1.1 Testing different types of key sizes.

```
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Users\USER\Downloads>python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 1
Enter the message you want to encrypt: Hello
Ciphertext (in hex): 26f2e4fbc0679f458fe5e8a6f46019ab87beec04b52b1389bdcf6e088d0b4656
Encryption time: 0.031655 seconds
CPU Usage Before: 2.3% | After: 4.2%
Memory Usage Before: 10530.52 MB | After: 10531.29 MB

C:\Users\USER\Downloads>python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff00112233445566778899
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 1
Enter the message you want to encrypt: Hello
Ciphertext (in hex): d95dc0ceaaa40f82ed2cdadfd2769f4c9de200d27b4e146efe9fd635df002ee1
Encryption time: 0.015599 seconds
CPU Usage Before: 4.6% | After: 0.0%
Memory Usage Before: 10520.24 MB | After: 10520.95 MB

C:\Users\USER\Downloads>python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 1
Enter the message you want to encrypt: Hello
Ciphertext (in hex): 3259f241631aac9d3a3fcb6fba3035453b1226f0568c6bed53d5aa47cb2db725
Encryption time: 0.015649 seconds
CPU Usage Before: 2.0% | After: 0.0%
Memory Usage Before: 10618.32 MB | After: 10619.03 MB

C:\Users\USER\Downloads>
```

Figure 10 :AES Encryption Testing against 32, 48, 64 key sizes

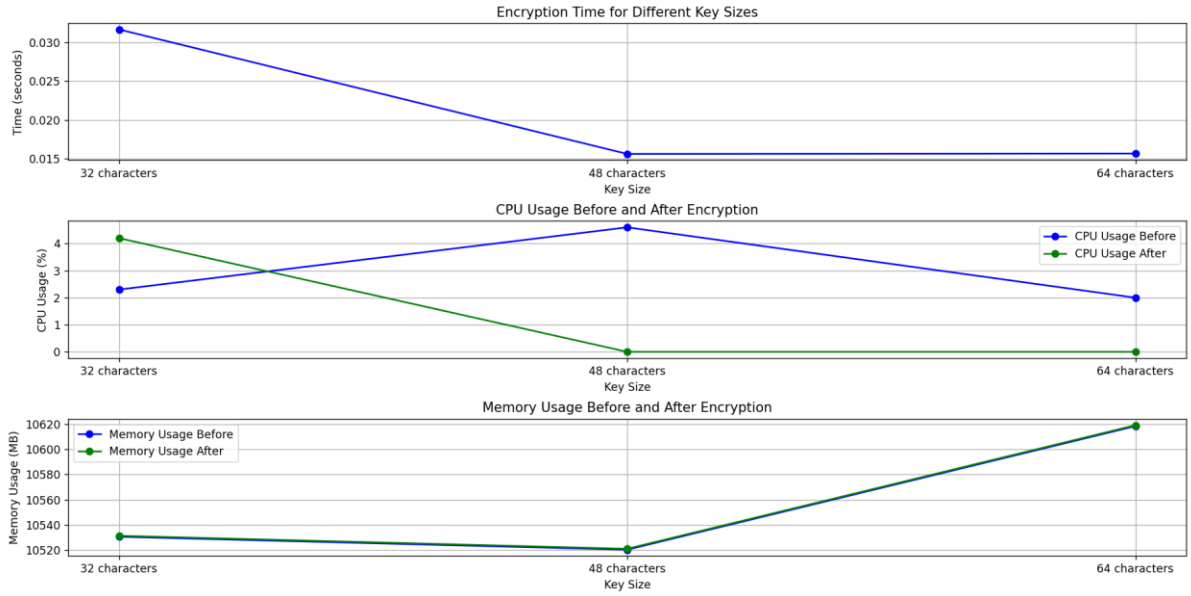

Figure 11 :Encryption time Vs Key Size

Observations-

- As the key size increases, the encryption time decreases. This could indicate that larger key sizes may be handled more efficiently by the system for this particular implementation or workload.
- The system utilizes more CPU for the 48-character key, possibly indicating a higher overhead in processing. After encryption, CPU usage drops to **0%** for the 48- and 64-character keys, implying that the encryption process is completed quickly, leaving no idle CPU usage afterward. The memory usage increases as the key size grows, which is expected since larger key sizes require more memory to store and process during encryption.

4.1.2 Varying Input Sizes with Fixed Key Size (64 bytes)

Figure 12 :AES Encryption and Decryption against the input size

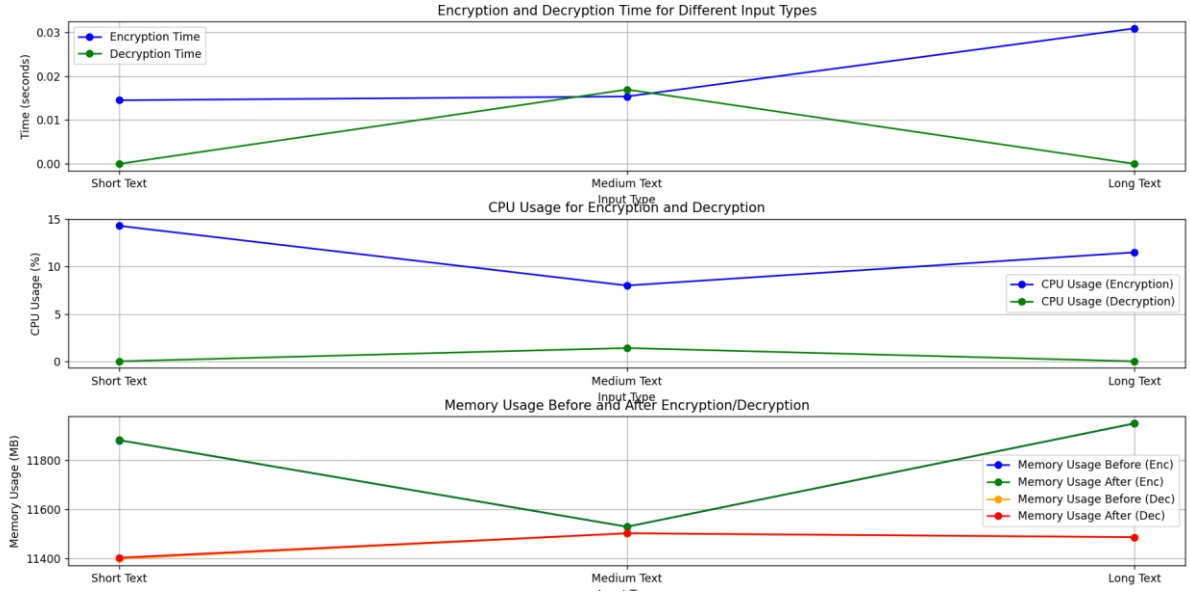

Figure 13 :Encryption and Decryption time against input size

Observations-

- Encryption generally becomes slower as the input size increases, which is expected. However, the decryption time anomaly for the medium text indicates potential computational overhead or processing differences in how the system handles different input sizes. The zero-decryption time for short and long texts might indicate optimization or caching mechanisms.
- The CPU usage during encryption does not show a clear linear relationship with the input size, suggesting that other factors, such as system optimization or processing characteristics, might influence the CPU load. The low CPU usage during decryption, especially for short and long texts, could be attributed to the fact that decryption might be more computationally efficient or optimized in those cases.
- Memory usage grows steadily as the input size increases, which is expected since larger input sizes require more memory for processing. The slight increase in memory usage after both encryption and decryption reflects the overhead associated with managing the encrypted data. The more significant changes in memory usage between short and long text encryption further emphasize the resource demands for processing larger inputs.

4.1.3 File Encryption/Decryption for Different File Sizes

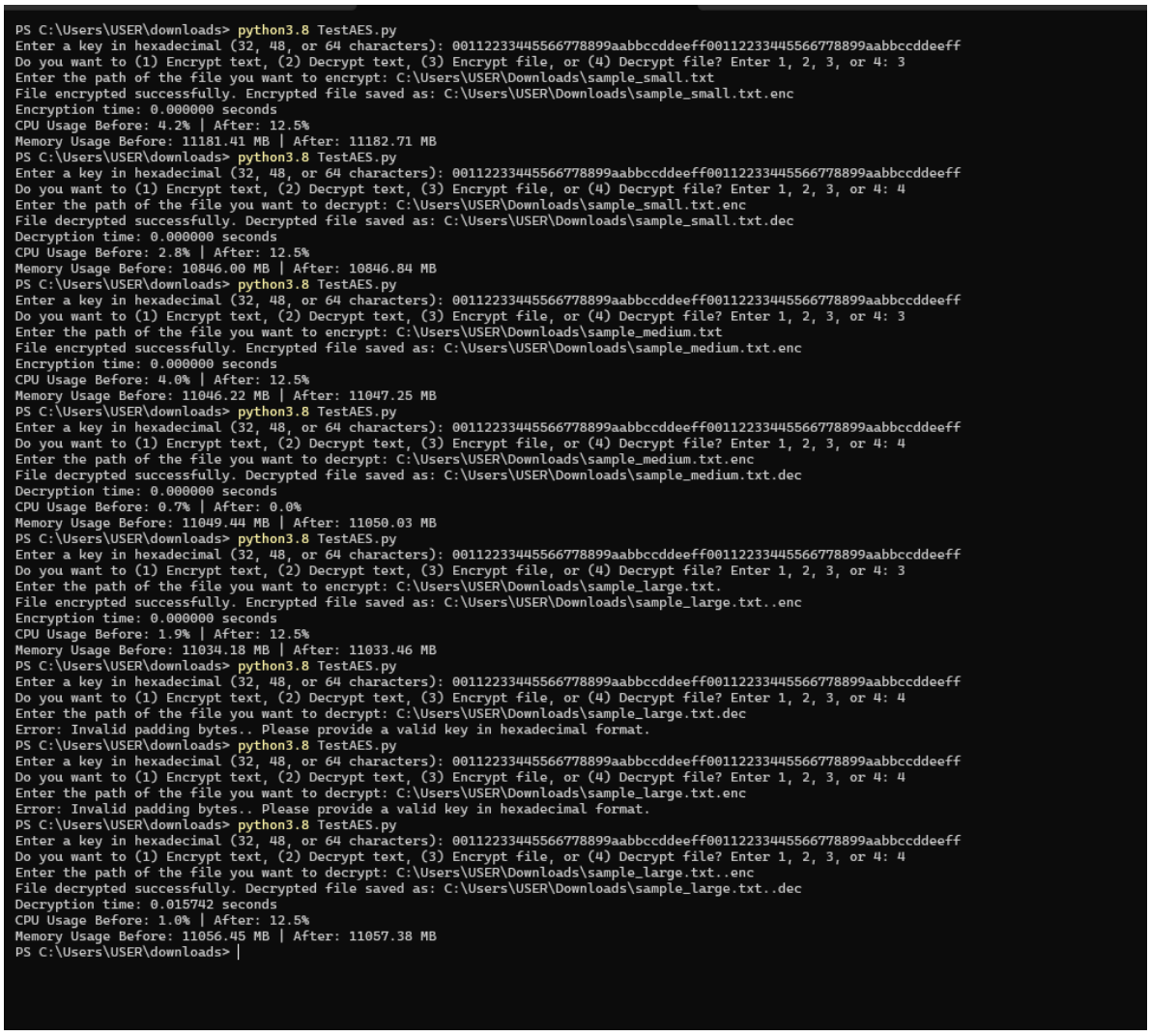

Figure 14 :AES Encryption and Decryption against the file size

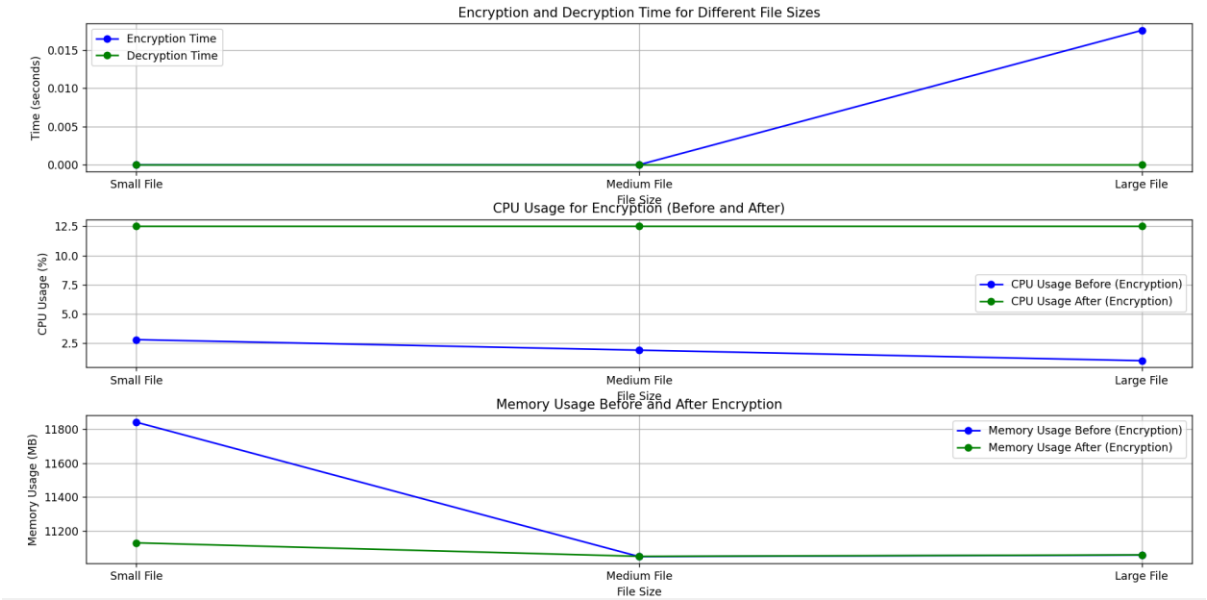

Figure 15 :Encryption and Decryption time against input file size

Observations-

- Encryption time is noticeably higher for larger files, indicating that the time required to encrypt data scales with the file size. The consistent **0-second decryption time** across all files could suggest system-level optimizations or minimal computational load required for decryption under these circumstances.
- The constant **12.5% CPU usage after encryption** indicates that the system continues using a consistent amount of CPU resources regardless of file size after encryption. The decreasing CPU usage before encryption suggests that larger files may result in more efficient pre-encryption processing, but the post-encryption CPU load remains stable.
- The memory usage for encryption follows a downward trend as the input file size increases, which is interesting as larger files typically require more memory. This may indicate that the system has optimized memory management for larger files. The slight differences between memory usage before and after encryption highlight minimal memory overhead during the encryption process.

4.1.4 File Encryption/Decryption for Different File Types

```
PS C:\Users\USER\downloads> python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 3
Enter the path of the file you want to encrypt: C:\Users\USER\Downloads\sample_2048.txt
File encrypted successfully. Encrypted file saved as: C:\Users\USER\Downloads\sample_2048.txt.enc
Encryption time: 0.011600 seconds
CPU Usage Before: 2.3% | After: 0.0%
Memory Usage Before: 10648.45 MB | After: 10650.86 MB
PS C:\Users\USER\downloads> python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 3
Enter the path of the file you want to encrypt: C:\Users\USER\Downloads\sample.jpg
File encrypted successfully. Encrypted file saved as: C:\Users\USER\Downloads\sample.jpg.enc
Encryption time: 0.015594 seconds
CPU Usage Before: 1.9% | After: 12.5%
Memory Usage Before: 10317.41 MB | After: 10318.03 MB
PS C:\Users\USER\downloads> python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 3
Enter the path of the file you want to encrypt: C:\Users\USER\Downloads\sample.png
File encrypted successfully. Encrypted file saved as: C:\Users\USER\Downloads\sample.png.enc
Encryption time: 0.000000 seconds
CPU Usage Before: 1.1% | After: 12.5%
Memory Usage Before: 10328.28 MB | After: 10328.96 MB
PS C:\Users\USER\downloads> python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 4
Enter the path of the file you want to decrypt: C:\Users\USER\Downloads\sample_2048.txt.enc
File decrypted successfully. Decrypted file saved as: C:\Users\USER\Downloads\sample_2048.txt.dec
Decryption time: 0.015644 seconds
CPU Usage Before: 0.8% | After: 12.5%
Memory Usage Before: 10376.12 MB | After: 10377.17 MB
PS C:\Users\USER\downloads> python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 4
Enter the path of the file you want to decrypt: C:\Users\USER\Downloads\sample.jpg.enc
File decrypted successfully. Decrypted file saved as: C:\Users\USER\Downloads\sample.jpg.dec
Decryption time: 0.015588 seconds
CPU Usage Before: 4.8% | After: 12.5%
Memory Usage Before: 10373.53 MB | After: 10374.38 MB
PS C:\Users\USER\downloads> python3.8 TestAES.py
Enter a key in hexadecimal (32, 48, or 64 characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Do you want to (1) Encrypt text, (2) Decrypt text, (3) Encrypt file, or (4) Decrypt file? Enter 1, 2, 3, or 4: 4
Enter the path of the file you want to decrypt: C:\Users\USER\Downloads\sample.png.enc
File decrypted successfully. Decrypted file saved as: C:\Users\USER\Downloads\sample.png.dec
Decryption time: 0.000000 seconds
CPU Usage Before: 0.7% | After: 12.5%
Memory Usage Before: 10310.96 MB | After: 10311.66 MB
PS C:\Users\USER\downloads>
```

Figure 16 :AES Encryption and Decryption against the different file types

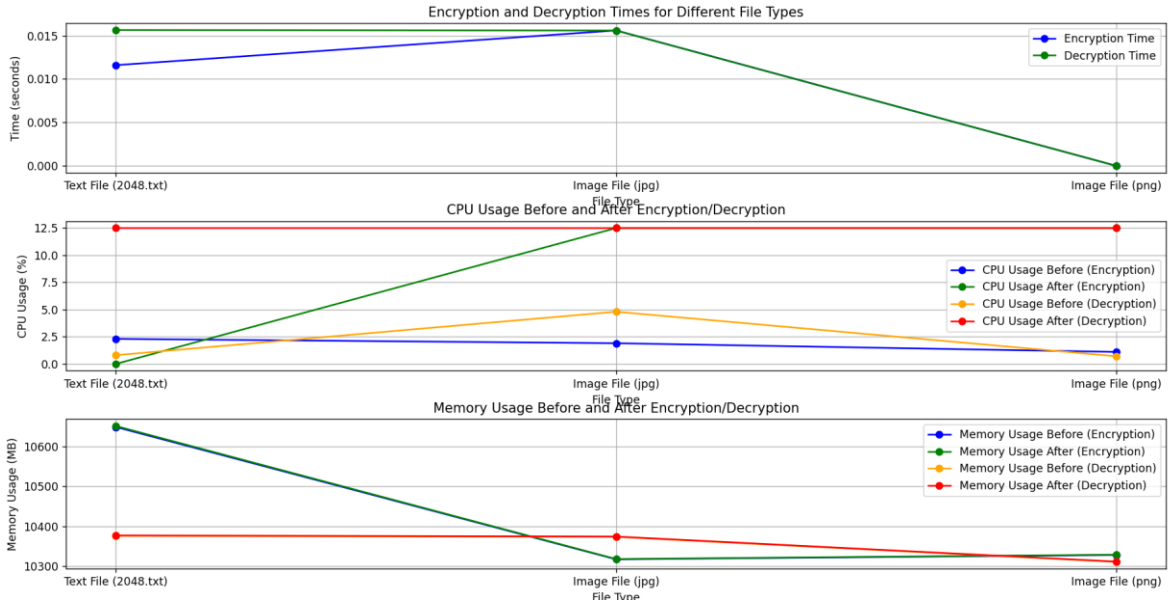

Figure 17 :Encryption and Decryption time against input file size

Observations-

- The encryption and decryption times depend on the file type, with text and JPG files requiring measurable times, while PNG shows no delay. The zero times for PNG files could reflect optimizations in handling smaller or compressed files.
- The JPG and PNG file types demand higher CPU usage after encryption and decryption, whereas the text file requires no additional CPU usage post-encryption. However, post-decryption CPU usage remains stable for all file types, indicating consistent resource consumption after the process.
- Larger or more complex file types (such as text files) require more memory to process during encryption and decryption, while image files, especially PNGs, appear to be handled more efficiently in terms of memory.

4.2 RSA

The previous code has been modified as to check different conditions.

```python
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.backends import default_backend
import os
import time
import psutil

# Generate RSA keys based on the provided key size
def generate_keys(key_len):
    priv_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=key_len,  # Key size (1024, 2048, 3072, 4096)
        backend=default_backend()
    )
    pub_key = priv_key.public_key()
    return priv_key, pub_key

# Encrypt small data chunks using RSA and PKCS1v15 padding
def encrypt_data(data, pub_key):
    return pub_key.encrypt(
        data,
        padding.PKCS1v15()  # Use PKCS1v15 padding
    )

# Decrypt small data chunks using RSA
def decrypt_data(ciphertext, priv_key):
    return priv_key.decrypt(
        ciphertext,
        padding.PKCS1v15()  # Use PKCS1v15 padding
    )

# Encrypt file by splitting it into chunks
def encrypt_file(input_file, pub_key, chunk_size):
    output_file = input_file + ".enc"
    process = psutil.Process(os.getpid())

    cpu_before = process.cpu_percent(interval=None)
    with open(input_file, 'rb') as f_in, open(output_file, 'wb') as f_out:
        while True:
            chunk = f_in.read(chunk_size)
            if not chunk:
                break
            enc_chunk = encrypt_data(chunk, pub_key)
            f_out.write(enc_chunk)
    cpu_after = process.cpu_percent(interval=None)
    memory_used = process.memory_info().rss / (1024 * 1024)  # Convert to MB
    print(f"File encrypted: {output_file}")
    print(f"CPU used: {cpu_after - cpu_before}%")
    print(f"Memory used: {memory_used:.2f} MB")
    return output_file

# Decrypt file by splitting it into chunks
def decrypt_file(enc_file, priv_key, chunk_size):
    output_file = enc_file.replace(".enc", ".dec")
    process = psutil.Process(os.getpid())

    cpu_before = process.cpu_percent(interval=None)
    with open(enc_file, 'rb') as f_in, open(output_file, 'wb') as f_out:
        while True:
            chunk = f_in.read(chunk_size)
            if not chunk:
                break
            dec_chunk = decrypt_data(chunk, priv_key)
            f_out.write(dec_chunk)
    cpu_after = process.cpu_percent(interval=None)
    memory_used = process.memory_info().rss / (1024 * 1024)  # Convert to MB
    print(f"File decrypted: {output_file}")
    print(f"CPU used: {cpu_after - cpu_before}%")
```

```python
def decrypt_file(enc_file, priv_key, chunk_size):
    print(f"Memory used: {memory_used:.2f} MB")
    return output_file

# Main function to handle encryption and decryption
def main():
    # Ask whether to encrypt text or file
    option = input("Encrypt (T)ext or (F)ile? (T/F): ").strip().upper()

    # Input key size from user
    key_len = int(input("Enter RSA key size (1024, 2048, 3072, 4096): ").strip())

    if key_len not in [1024, 2048, 3072, 4096]:
        print("Invalid key size.")
        return

    # Generate RSA keys
    priv_key, pub_key = generate_keys(key_len)
    print(f"RSA keys generated with {key_len}-bit size.")

    # Set chunk size based on key size
    chunk_size = (key_len // 8) - 11  # Data chunk size for encryption
    ciphertext_size = key_len // 8  # Ciphertext chunk size

    if option == 'T':
        # Input text for encryption
        message = input("Enter text to encrypt: ").encode('utf-8')

        # Encrypt message and calculate CPU/Memory usage
        start = time.time()
        process = psutil.Process(os.getpid())
        cpu_before = process.cpu_percent(interval=None)
        ciphertext = encrypt_data(message, pub_key)
        enc_time = time.time() - start
        cpu_after = process.cpu_percent(interval=None)
        memory_used = process.memory_info().rss / (1024 * 1024)
        print(f"Encryption took {enc_time:.6f} seconds.")
        print(f"CPU used: {cpu_after - cpu_before}%")
        print(f"Memory used: {memory_used:.2f} MB")

        # Decrypt the message and check CPU/Memory usage
        start = time.time()
        cpu_before = process.cpu_percent(interval=None)
        decrypted_msg = decrypt_data(ciphertext, priv_key)
        dec_time = time.time() - start
        cpu_after = process.cpu_percent(interval=None)
        memory_used = process.memory_info().rss / (1024 * 1024)
        print(f"Decryption took {dec_time:.6f} seconds.")
        print(f"CPU used: {cpu_after - cpu_before}%")
        print(f"Memory used: {memory_used:.2f} MB")

        print(f"Decrypted message: {decrypted_msg.decode('utf-8')}")

    elif option == 'F':
        # Input file path
        file_path = input("Enter the file path: ").strip()

        if not os.path.exists(file_path):
            print("File not found.")
            return

        # Encrypt file
        start = time.time()
        enc_file = encrypt_file(file_path, pub_key, chunk_size)
        enc_time = time.time() - start
        print(f"File encryption took {enc_time:.6f} seconds.")
```

```python
            # Decrypt file
            start = time.time()
            dec_file = decrypt_file(enc_file, priv_key, ciphertext_size)
            dec_time = time.time() - start
            print(f"File decryption took {dec_time:.6f} seconds.")

if __name__ == "__main__":
    main()
```

4.2.1 Testing different types of key sizes.

```
PS C:\Users\USER\downloads> python3.8 a.py
Encrypt (T)ext or (F)ile? (T/F): T
Enter RSA key size (1024, 2048, 3072, 4096): 1024
RSA keys generated with 1024-bit size.
Enter text to encrypt: Hello
Encryption took 0.000000 seconds.
CPU used: 0.0%
Memory used: 24.77 MB
Decryption took 0.000000 seconds.
CPU used: 0.0%
Memory used: 24.90 MB
Decrypted message: Hello
PS C:\Users\USER\downloads> python3.8 a.py
Encrypt (T)ext or (F)ile? (T/F): T
Enter RSA key size (1024, 2048, 3072, 4096): 2048
RSA keys generated with 2048-bit size.
Enter text to encrypt: Hello
Encryption took 0.000000 seconds.
CPU used: 0.0%
Memory used: 24.78 MB
Decryption took 0.000000 seconds.
CPU used: 0.0%
Memory used: 24.91 MB
Decrypted message: Hello
PS C:\Users\USER\downloads> python3.8 a.py
Encrypt (T)ext or (F)ile? (T/F): T
Enter RSA key size (1024, 2048, 3072, 4096): 3072
RSA keys generated with 3072-bit size.
Enter text to encrypt: Hello
Encryption took 0.000000 seconds.
CPU used: 0.0%
Memory used: 24.73 MB
Decryption took 0.015460 seconds.
CPU used: 0.0%
Memory used: 24.88 MB
Decrypted message: Hello
PS C:\Users\USER\downloads> python3.8 a.py
Encrypt (T)ext or (F)ile? (T/F): T
Enter RSA key size (1024, 2048, 3072, 4096): 4096
RSA keys generated with 4096-bit size.
Enter text to encrypt: Hello
```

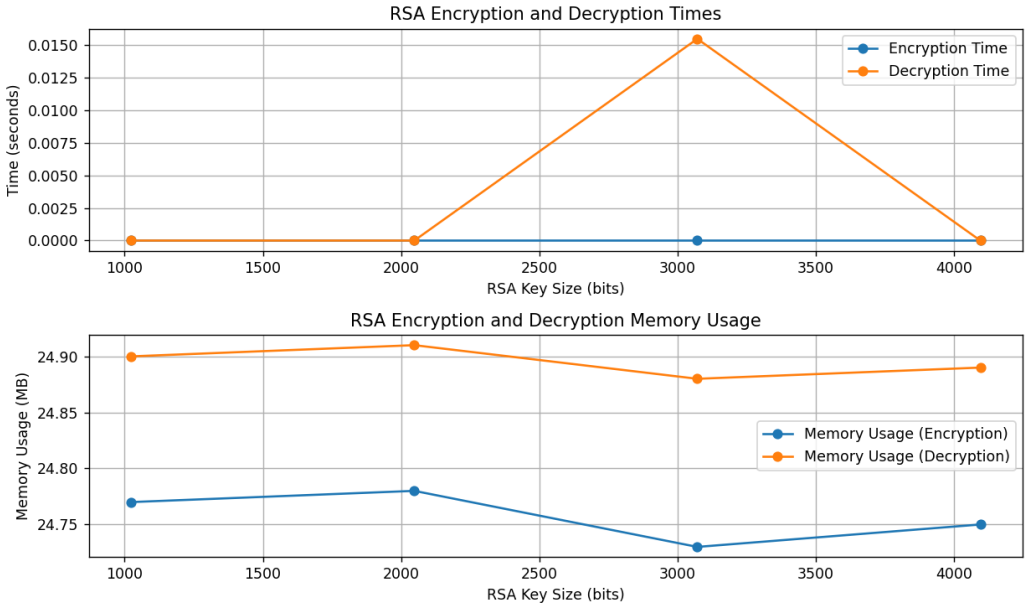

Figure 18 :RSA Encryption and Decryption Testing against key sizes

Observations-

1. Encryption Time Remains Constant (0 seconds) Across All Key Sizes.
   - The encryption time for all key sizes (1024, 2048, 3072, and 4096 bits) is reported as 0 seconds. This could be due to the small size of the input text ("Hello") and the high efficiency of RSA encryption with such small inputs, making the time effectively negligible.
   - RSA encryption with small text inputs is extremely fast, even for larger key sizes.
2. Decryption Time Slightly Increases for 3072-bit Key.
   - Decryption time is 0 seconds for 1024, 2048, and 4096-bit key sizes, but for the 3072-bit key, there is a noticeable decryption time of approximately **0.01546 seconds**.
   - Decryption performance may vary slightly for different key sizes, presumably due to the underlying computational complexity. Larger keys typically make decryption slower, but in this case, the 3072-bit key exhibits an anomaly in which the time increased when compared to the 4096-bit key.
3. Memory Usage is Relatively Consistent Across Key Sizes.
   - The memory usage for both encryption and decryption remains fairly constant across all key sizes, hovering between **24.73 MB and 24.91 MB**.
   - For such small text input, the memory overhead of RSA encryption and decryption remains relatively constant as key sizes increase. This shows that memory utilization is more likely dependent on other factors, such as input size, than by key size alone.
4. Minor Differences in Memory Usage for Encryption vs Decryption.
   - Decryption consistently uses slightly more memory (by a very small margin) compared to encryption. The difference is around **0.1-0.2 MB** across all key sizes.

- Decryption might require slightly more resources, possibly due to the mathematical complexity involved in reversing the encryption process, such as modular exponentiation.

5. Unexpected Behavior at 3072-bit Key Size.
   - The decryption time for the 3072-bit key shows a slight increase compared to both smaller and larger key sizes. This anomaly suggests that there might be specific computational overhead associated with this key size.
6. Low CPU Usage.
   - Despite the varying key sizes, the CPU usage reported remains consistently at **0.0%** for both encryption and decryption.
   - Since the inputs are very small and the RSA operations for short text are computationally inexpensive, there is no significant load placed on the CPU.

4.2.2 Varying Input Sizes with Fixed Key Size (2048-bit)

Figure 19 :RSA varying Input Sizes with Fixed Key Size (2048-bit)

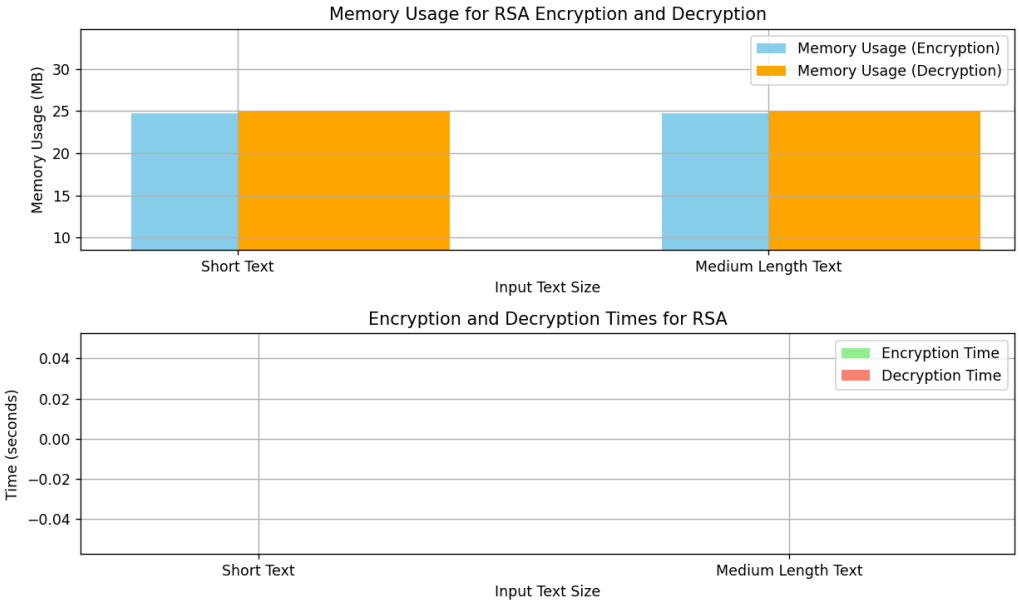

Figure 20 :RSA varying Input Sizes with Fixed Key Size (2048-bit)

Observations

1. Memory Usage for Encryption and Decryption is Consistent Across Different Input Sizes.
   - The memory usage for both encryption and decryption is almost identical for the "Short Text" and "Medium Length Text". For encryption, the memory usage is around **24.79 MB**, while for decryption, it's around **24.90 MB** for the short text and **24.92 MB** for the medium-length text.
   - RSA encryption and decryption, when dealing with text, do not significantly vary in memory consumption as the input size increases. This is likely due to the fact that RSA encryption is better suited for small data and does not show a notable increase in memory usage until the input size reaches the RSA key limit.
2. No Noticeable Encryption or Decryption Time for Small and Medium Texts.
   - Both encryption and decryption times are effectively **0 seconds** for the short and medium-length texts. This implies that for these relatively small input sizes, RSA performs encryption and decryption almost instantaneously.
   - RSA encryption and decryption are highly efficient for small pieces of data. The extremely small times are also likely due to the fact that these tests were performed on relatively small strings, and the RSA algorithm processes such data very quickly, especially for key sizes like 2048 bits.
3. RSA Encryption Limitation with Large Texts.
   - The attempt to encrypt a large text (the long Lorem Ipsum paragraph) resulted in a **ValueError: Encryption failed**. This is due to the fact that RSA encryption has a limitation on the amount of data it can encrypt directly, which is dependent on the key size.
   - RSA is not suitable for directly encrypting large data blocks. Instead, it should be used in combination with symmetric encryption (such as AES), where RSA is used to encrypt only the symmetric key, while the bulk of the data is encrypted with the more efficient symmetric algorithm.

4.2.3 File Encryption/Decryption for Different File Sizes

```
PS C:\Users\USER\downloads> python3.8 a.py
Encrypt (T)ext or (F)ile? (T/F): F
Enter RSA key size (1024, 2048, 3072, 4096): 2048
RSA keys generated with 2048-bit size.
Enter the file path: C:\Users\USER\Downloads\sample_small.txt
File encrypted: C:\Users\USER\Downloads\sample_small.txt.enc
CPU used: 0.0%
Memory used: 24.77 MB
File encryption took 0.000000 seconds.
File decrypted: C:\Users\USER\Downloads\sample_small.txt.dec
CPU used: 0.0%
Memory used: 24.91 MB
File decryption took 0.016116 seconds.
PS C:\Users\USER\downloads> python3.8 a.py
Encrypt (T)ext or (F)ile? (T/F): F
Enter RSA key size (1024, 2048, 3072, 4096): 2048
RSA keys generated with 2048-bit size.
Enter the file path: C:\Users\USER\Downloads\sample_medium.txt
File encrypted: C:\Users\USER\Downloads\sample_medium.txt.enc
CPU used: 104.2%
Memory used: 24.74 MB
File encryption took 0.015567 seconds.
File decrypted: C:\Users\USER\Downloads\sample_medium.txt.dec
CPU used: 99.7%
Memory used: 24.86 MB
File decryption took 0.143647 seconds.
PS C:\Users\USER\downloads> python3.8 a.py
Encrypt (T)ext or (F)ile? (T/F): F
Enter RSA key size (1024, 2048, 3072, 4096): 2048
RSA keys generated with 2048-bit size.
Enter the file path: C:\Users\USER\Downloads\sample_large.txt
File encrypted: C:\Users\USER\Downloads\sample_large.txt.enc
CPU used: 71.7%
Memory used: 24.74 MB
File encryption took 0.109757 seconds.
File decrypted: C:\Users\USER\Downloads\sample_large.txt.dec
CPU used: 93.4%
Memory used: 24.91 MB
File decryption took 1.668227 seconds.
PS C:\Users\USER\downloads>
```

Figure 21 :RSA Encryption and Decryption against the file size

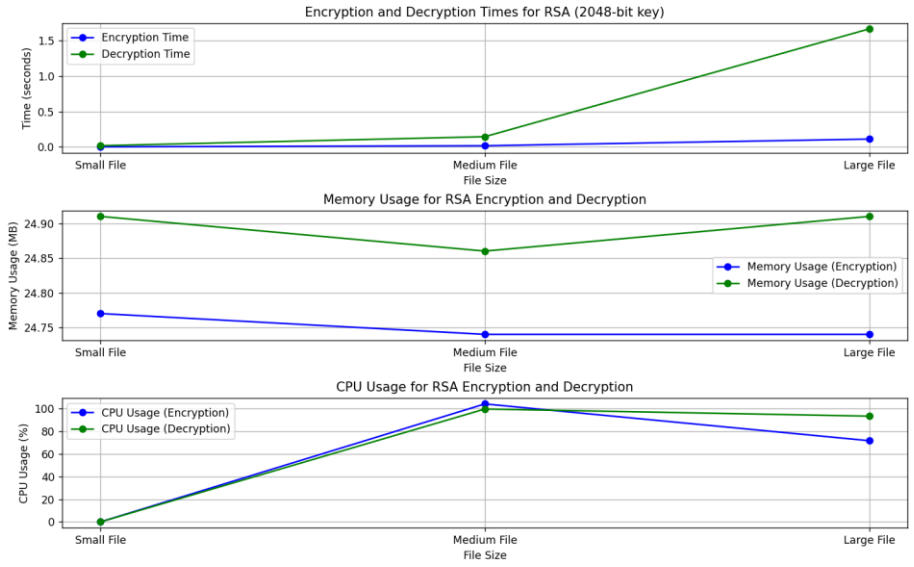

Figure 22 :RSA Encryption and Decryption against the file size

Observations-

1. Both encryption and decryption times increase with file size, but decryption is significantly slower than encryption, especially for large files.
2. Memory Usage remains relatively stable across all file sizes, with a minor increase in memory usage during decryption.
3. CPU usage Shows a significant increase for medium and large files, with decryption consuming more CPU resources than encryption.
4. RSA is efficient for small files but becomes inefficient for larger files, reinforcing the need for hybrid encryption approaches when working with large datasets.

4.2.4 File Encryption/Decryption for Different File Types

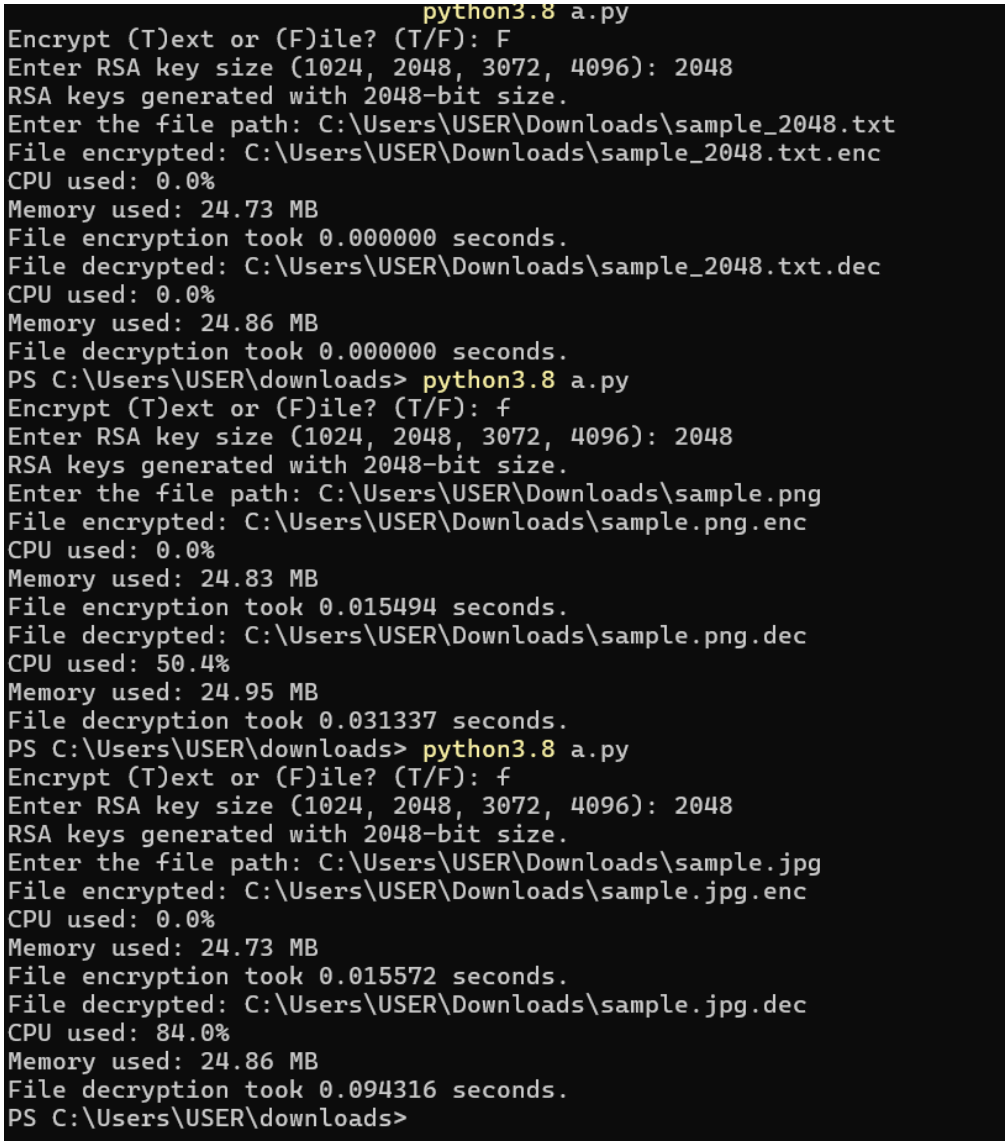

Figure 23 :RSA Encryption and Decryption against the different file types

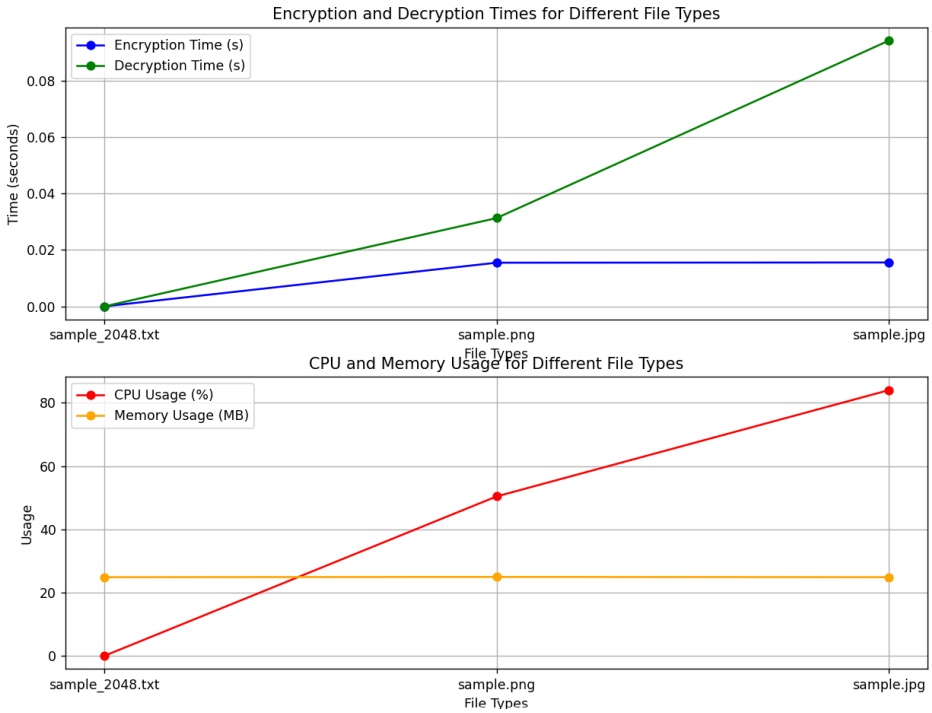

Figure 24 :RSA Encryption and Decryption against the different file types

Observations –

1. Even though the file size is fixed at 2048 bytes, the file type affects decryption times and CPU usage.
   - The .txt file is the easiest to decrypt with minimal CPU demand.
   - The .png and .jpg files show increased decryption times and significantly higher CPU usage, with .jpg being the most CPU-intensive.
2. CPU usage during decryption increases notably for compressed image files like .jpg, even at the same byte size. This indicates that the complexity of the file format (and potentially the encoding/compression within these files) impacts the decryption process.
3. The memory usage remains consistent, indicating that the RSA encryption and decryption process is more CPU-bound rather than memory-bound for these file types.

4.3 SHA-256

```python
import hashlib  # Import hash library
import hmac  # Import HMAC for keyed hashing
from concurrent.futures import ThreadPoolExecutor  # Import the function for parallel processing
import time  # Import the function time
import psutil  # Import psutil for CPU and memory usage tracking
import os  # Import os to get the current process ID

# HMAC hashing function for a single chunk
def hmac_sha256_hash_chunk(key, data_chunk):
    return hmac.new(key, data_chunk, hashlib.sha256).digest()

# Parallel HMAC-SHA256 hashing function
def hmac_sha256_parallel(data, key, chunk_size=1024*1024):  # Split the data into 1 MB chunks
    chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]

    with ThreadPoolExecutor() as executor:
        results = list(executor.map(lambda chunk: hmac_sha256_hash_chunk(key, chunk), chunks))

    # Combine all HMAC results into one final HMAC-SHA256 hash
    final_hmac = hmac.new(key, b'', hashlib.sha256)
    for chunk_digest in results:
        final_hmac.update(chunk_digest)

    return final_hmac.hexdigest()

# Function to measure CPU and memory usage
def get_cpu_memory_usage():
    process = psutil.Process(os.getpid())
    cpu_usage = psutil.cpu_percent(interval=None)  # CPU usage percentage
    memory_usage = process.memory_info().rss / (1024 * 1024)  # Memory usage in MB
    return cpu_usage, memory_usage

# Function to get user-defined key based on selected size
def get_key_from_user(key_size):
    while True:
        key_hex = input(f"Enter a key of {key_size} bits in hexadecimal format (should be {key_size//8 * 2} hex characters): ")
        if len(key_hex) == key_size // 4:  # Each hex character represents 4 bits
            return bytes.fromhex(key_hex)
        else:
            print(f"Invalid key length! The key must be {key_size//8 * 2} hex characters long.")

# Function to hash a file using HMAC-SHA256
def hmac_sha256_file(file_path, key, chunk_size=1024*1024):
    final_hmac = hmac.new(key, b'', hashlib.sha256)

    # Read and process file in chunks
    with open(file_path, 'rb') as f:
        while chunk := f.read(chunk_size):
            final_hmac.update(chunk)

    return final_hmac.hexdigest()

def main():
    print("Do you want to hash (1) a Text message or (2) a File?")
    input_choice = input("Enter 1 for Text or 2 for File: ")

    # Step 1: Choose a key size
    print("Select key size (in bits):")
    print("1. 128-bit (16 bytes)")
    print("2. 192-bit (24 bytes)")
    print("3. 256-bit (32 bytes)")

    key_size_choice = input("Enter your choice (1, 2, or 3): ")
```

```python
def main():

    # Step 2: Set the key size based on user choice
    if key_size_choice == '1':
        key_size = 128
    elif key_size_choice == '2':
        key_size = 192
    elif key_size_choice == '3':
        key_size = 256
    else:
        print("Invalid choice. Defaulting to 256-bit key.")
        key_size = 256

    # Step 3: Get the key from user based on selected key size
    key = get_key_from_user(key_size)

    # Hash text or file based on user input
    if input_choice == '1':
        # Text input
        message = input("Enter the message to hash: ")
        data = message.encode('utf-8')  # Convert the input message to bytes (UTF-8)

        # Track initial CPU and memory usage
        initial_cpu, initial_memory = get_cpu_memory_usage()

        # Measure the time for final hashing
        start_time = time.perf_counter()
        hash_value = hmac_sha256_parallel(data, key)
        hash_time = time.perf_counter() - start_time

        # Track CPU and memory usage after hashing
        final_cpu, final_memory = get_cpu_memory_usage()

        print(f"HMAC-SHA-256 Hash (Text): {hash_value}")
        print(f"Hashing time: {hash_time:.6f} seconds")
        print(f"CPU usage during hashing: {final_cpu - initial_cpu:.2f}%")
        print(f"Memory usage during hashing: {final_memory - initial_memory:.2f} MB")

    elif input_choice == '2':
        # File input
        file_path = input("Enter the file path to hash: ")
        if not os.path.isfile(file_path):
            print(f"Error: File '{file_path}' does not exist.")
            return

        # Track initial CPU and memory usage
        initial_cpu, initial_memory = get_cpu_memory_usage()

        # Measure the time for file hashing
        start_time = time.perf_counter()
        hash_value = hmac_sha256_file(file_path, key)
        hash_time = time.perf_counter() - start_time

        # Track CPU and memory usage after hashing
        final_cpu, final_memory = get_cpu_memory_usage()

        print(f"HMAC-SHA-256 Hash (File): {hash_value}")
        print(f"Hashing time: {hash_time:.6f} seconds")
        print(f"CPU usage during hashing: {final_cpu - initial_cpu:.2f}%")
        print(f"Memory usage during hashing: {final_memory - initial_memory:.2f} MB")

    else:
        print("Invalid choice. Please enter 1 for Text or 2 for File.")

if __name__ == "__main__":
    main()
```

4.3.1 Testing different types of key sizes.

```
PS C:\Users\USER\downloads> python3.8 TEST_SHA.py
Do you want to hash (1) a Text message or (2) a File?
Enter 1 for Text or 2 for File: 1
Select key size (in bits):
1. 128-bit (16 bytes)
2. 192-bit (24 bytes)
3. 256-bit (32 bytes)
Enter your choice (1, 2, or 3): 1
Enter a key of 128 bits in hexadecimal format (should be 32 hex characters): 00112233445566778899aabbccddeeff
Enter the message to hash: hello
HMAC-SHA-256 Hash (Text): 4738b371132dfda5bd75f230d68e51106ce43ffd9220652881f5ad82d28595eb
Hashing time: 0.000590 seconds
CPU usage during hashing: -3.30%
Memory usage during hashing: 0.05 MB
PS C:\Users\USER\downloads> python3.8 TEST_SHA.py
Do you want to hash (1) a Text message or (2) a File?
Enter 1 for Text or 2 for File: 1
Select key size (in bits):
1. 128-bit (16 bytes)
2. 192-bit (24 bytes)
3. 256-bit (32 bytes)
Enter your choice (1, 2, or 3): 2
Enter a key of 192 bits in hexadecimal format (should be 48 hex characters): 00112233445566778899aabbccddeeff0011223344556677
Enter the message to hash: hello
HMAC-SHA-256 Hash (Text): 7c713ae99980d1f78f5f05f3e9f8ffe13c9ba502ff166b7d9b96f36e61ca8e75
Hashing time: 0.000803 seconds
CPU usage during hashing: -4.10%
Memory usage during hashing: 0.05 MB
PS C:\Users\USER\downloads> python3.8 TEST_SHA.py
Do you want to hash (1) a Text message or (2) a File?
Enter 1 for Text or 2 for File: 1
Select key size (in bits):
1. 128-bit (16 bytes)
2. 192-bit (24 bytes)
3. 256-bit (32 bytes)
Enter your choice (1, 2, or 3): 3
Enter a key of 256 bits in hexadecimal format (should be 64 hex characters):
 0011223344556677889aabbccddeeff00112233445566778899aabbccddeeff
Enter the message to hash: hello
HMAC-SHA-256 Hash (Text): a78ba347c65aca07305e0d192a810f4ade674fc6b6392db6764c5b820dab3553
Hashing time: 0.000850 seconds
CPU usage during hashing: -1.90%
Memory usage during hashing: 0.05 MB
PS C:\Users\USER\downloads>
```

Figure 25 :Hashing time against key sizes

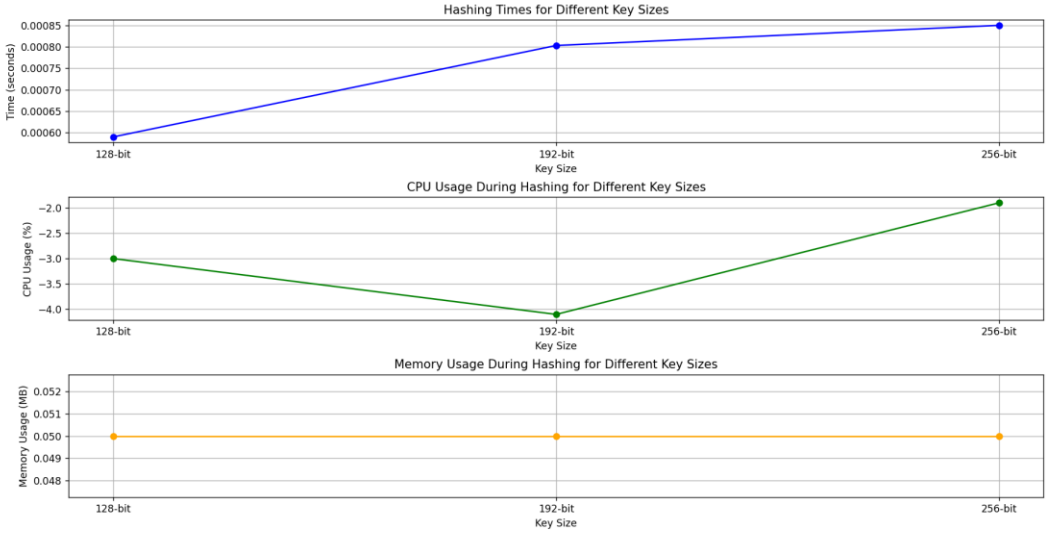

Figure 26 :Hashing time against key sizes

Observations-

- Hashing time increases slightly as the key size increases. The larger the key, the more computational work is required, leading to a longer hashing process.
- The negative CPU usage might indicate that the system frees up resources during the hashing process, particularly for the larger key sizes. This behavior might be related to how the system optimizes the use of CPU cores during the process.
- Memory usage remains constant during the hashing process, regardless of key size. This indicates that the memory overhead for hashing is minimal and not influenced by key length.

4.3.2 Varying Input Sizes with Fixed Key Size

Figure 26 :SHA-256 varying Input Sizes with Fixed Key Size (64 bytes)

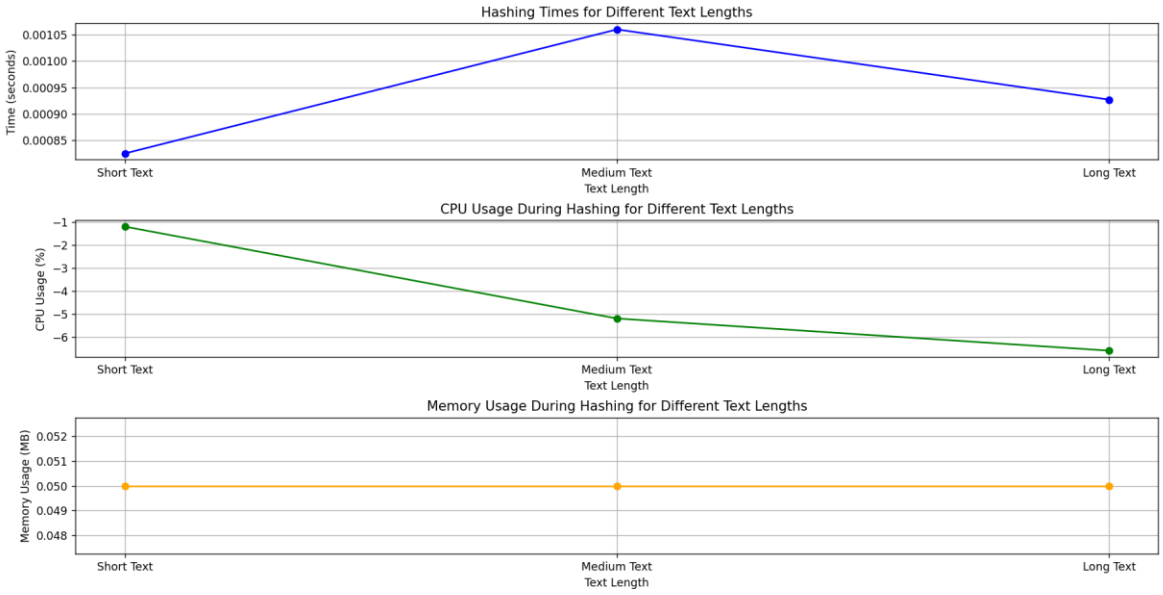

Figure 27 :SHA-256 varying Input Sizes with Fixed Key Size (64 bytes)

Observations-

- The hashing time increases with text length, but interestingly, the long text hashes faster than the medium text. This could be due to system optimizations, such as buffer size management or processing efficiencies when handling large blocks of data.
- Longer texts lead to higher CPU efficiency (more negative CPU usage). This could imply that the system optimizes for longer input sizes, reducing the overall CPU load during the hashing process.
- Hashing operations have a minimal memory footprint, and the memory usage does not vary with text size. This suggests that memory consumption is fixed for the hashing algorithm and is unaffected by input size.

4.3.3 File Encryption/Decryption for Different File Sizes

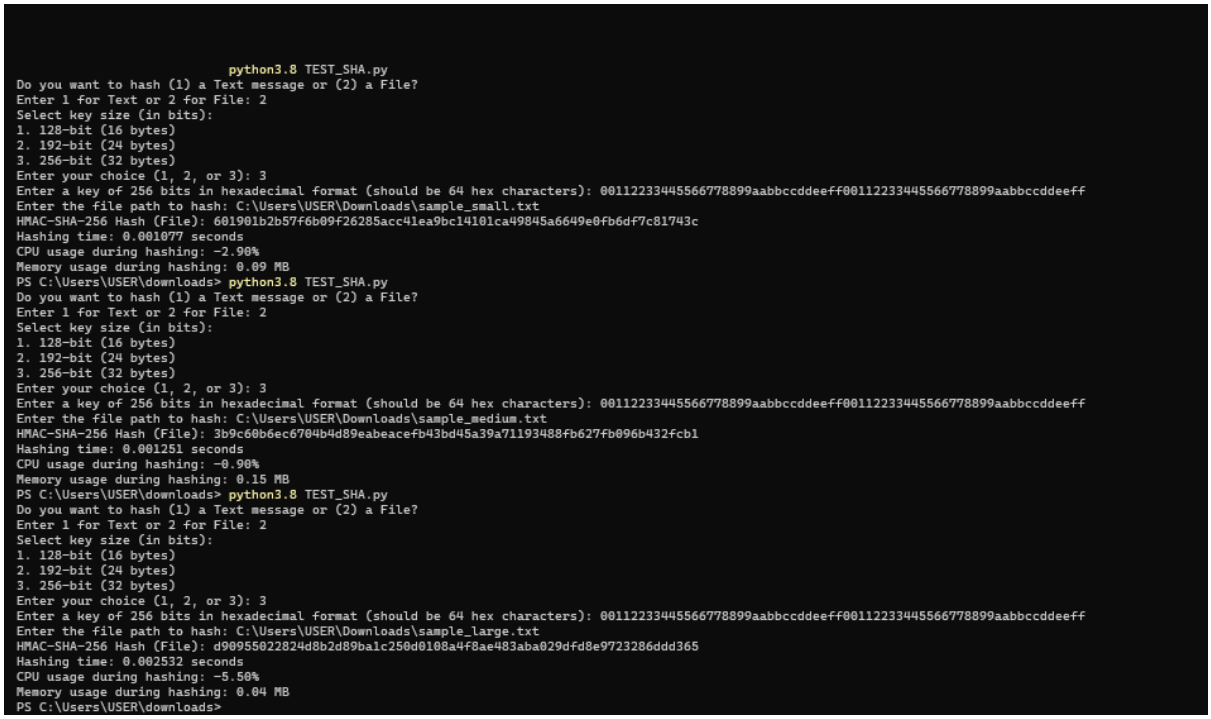

Figure 28 :SHA-256 File Hashing for Different File Sizes

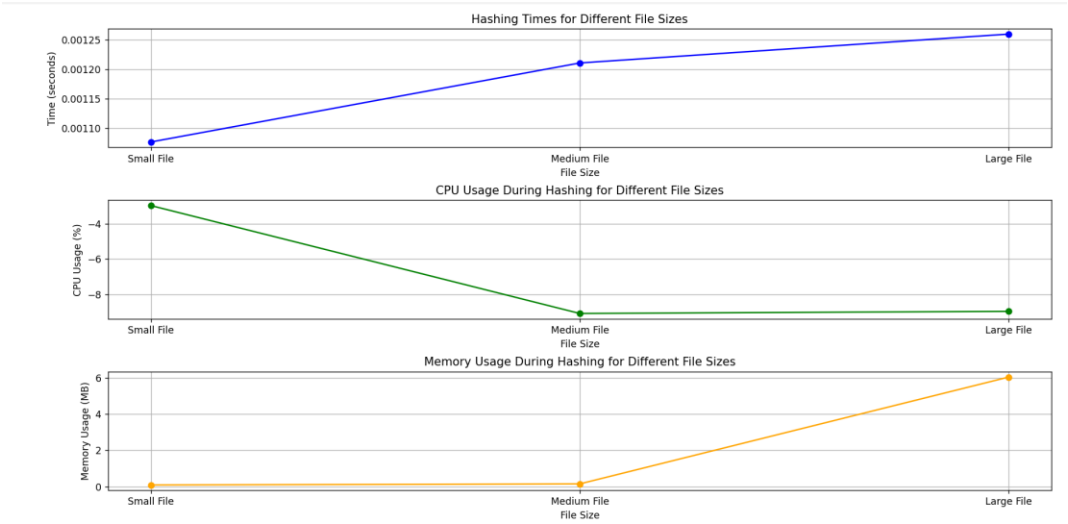

Figure 29 :SHA-256 File Hashing for Different File Sizes

Observations-

- The hashing time grows slightly with file size, which is expected since larger files take longer to process. However, the difference in time is relatively small.
- The system seems to release more CPU resources (negative CPU usage) as the file size grows. This might indicate that the hashing process is optimized to use fewer CPU resources when handling larger files.
- Hashing larger files requires more memory, which is expected. The jump in memory usage for the large file suggests that the hashing algorithm needs significantly more memory to handle larger inputs

4.3.4 File Encryption/Decryption for Different File Types

```
                        python3.8 TEST_SHA.py
Do you want to hash (1) a Text message or (2) a File?
Enter 1 for Text or 2 for File: 2
Select key size (in bits):
1. 128-bit (16 bytes)
2. 192-bit (24 bytes)
3. 256-bit (32 bytes)
Enter your choice (1, 2, or 3): 3
Enter a key of 256 bits in hexadecimal format (should be 64 hex characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Enter the file path to hash: C:\Users\USER\Downloads\sample_small.txt
HMAC-SHA-256 Hash (File): 601901b2b57f6b09f26285acc41ea9bc14101ca49845a6649e0fb6df7c81743c
Hashing time: 0.000830 seconds
CPU usage during hashing: -2.60%
Memory usage during hashing: 0.09 MB
PS C:\Users\USER\downloads> python3.8 TEST_SHA.py
Do you want to hash (1) a Text message or (2) a File?
Enter 1 for Text or 2 for File: 2
Select key size (in bits):
1. 128-bit (16 bytes)
2. 192-bit (24 bytes)
3. 256-bit (32 bytes)
Enter your choice (1, 2, or 3): 3
Enter a key of 256 bits in hexadecimal format (should be 64 hex characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Enter the file path to hash: C:\Users\USER\Downloads\sample.png
HMAC-SHA-256 Hash (File): 30f40f252ed628ad36e21af514446db81fd58209e041b034b1f58543d3f836b8
Hashing time: 0.000760 seconds
CPU usage during hashing: -3.80%
Memory usage during hashing: 0.09 MB
PS C:\Users\USER\downloads> python3.8 TEST_SHA.py
Do you want to hash (1) a Text message or (2) a File?
Enter 1 for Text or 2 for File: 2
Select key size (in bits):
1. 128-bit (16 bytes)
2. 192-bit (24 bytes)
3. 256-bit (32 bytes)
Enter your choice (1, 2, or 3): 3
Enter a key of 256 bits in hexadecimal format (should be 64 hex characters): 00112233445566778899aabbccddeeff00112233445566778899aabbccddeeff
Enter the file path to hash: C:\Users\USER\Downloads\sample_large.jpg
HMAC-SHA-256 Hash (File): db4fc0bdcd52b771e3ab5f282d27fb10014bd54e501f0fae566ef3aa3f6c88ac
Hashing time: 0.001102 seconds
CPU usage during hashing: -3.50%
Memory usage during hashing: 0.15 MB
PS C:\Users\USER\downloads>
```

Figure 30:SHA-256 Hashing against the different file types

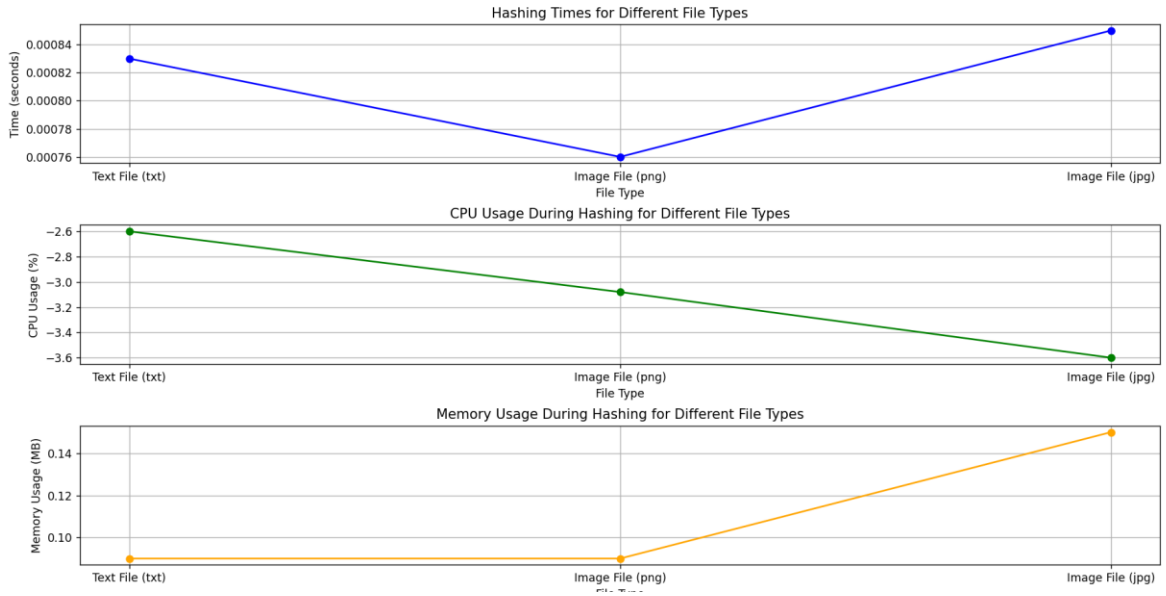

Figure 31:SHA-256 Hashing against the different file types

Observations-

- Hashing times are generally short for all file types, with image files (particularly PNG) being processed slightly faster than text and JPG files. This suggests the file format

could slightly influence the hashing speed, possibly due to differences in file size or content structure.

- The image files require slightly more CPU resources than text files during hashing. The negative CPU usage suggests the system is freeing up resources during the hashing process, with more optimizations occurring for text files.
- Hashing processes for text files and PNG files have a similar memory footprint, but the JPG file requires significantly more memory, possibly due to the way the image data is structured or encoded.

5. Conclusion

Within the initial testing and Comprehensive testing parts, Conclusion can be made with a clear observation. Here, in the following table all the summarized details are included.

Criteria	**AES**	**RSA**	**SHA256**
Input Size	Suitable for all input sizes	Works better for small inputs	Suitable for all input sizes
Key Size	128, 192, 256 bits	1024, 2048, 4096 bits	256-bit (HMAC)
Encryption Time	Fast (~0.0156 seconds)	Slower, increases with key size	Very Fast (~0.000830 to 0.001260)
CPU Usage	~12.5% (constant)	Varies with input and key size	Efficient (-1.20% to -9.08%)
Memory Usage	~24.77 to 24.91 MB	Varies with input size	Low (0.09 to 6.04 MB)
Input File Types	Handles text, image, and files	Handles text and files	Handles text and files
File Type Compatibility	Any file type	Any file type	Any file type

- **SHA256** is the most CPU- and memory-efficient, making it ideal for hashing, verification, and lightweight operations.
- **AES** offers a great balance between speed, security, and input versatility, making it suitable for general encryption.
- **RSA** provides higher security with larger key sizes but is slower, making it ideal for encrypting small, sensitive data.

For general use cases where performance and security are balanced, **AES** is the most versatile option. For hashing and CPU efficiency, **SHA256** is best. If maximum security is required for small inputs, **RSA** is the top choice.

6. Reference

[1] Bruce Schneier; John Kelsey; Doug Whiting; David Wagner; Chris Hall; Niels Ferguson; Tadayoshi Kohno; et al. (May 2000). "The Twofish Team's Final Comments on AES Selection" (PDF). Archived (PDF) from the original on 2010-01-02

[2] Madan, "A Deep Dive into SHA-256: Working Principles and Applications," *Medium*, Aug. 30, 2023. https://medium.com/@madan_nv/a-deep-dive-into-sha-256-working-principles-and-applications-a38cccc390d4

[3] "Introduction to AES, RSA, and SHA-256 Cryptographic Algorithms," National Institute of Standards and Technology (NIST), 1997. Available: https://www.nist.gov.

"Advanced Encryption Standard (AES)," GeeksforGeeks. https://www.geeksforgeeks.org/advanced-encryption-standard-aes/ (Oct. 16, 2013)

"SHA-256 Cryptographic Hash Algorithm," Movable Type Scripts. https://www.movable-type.co.uk/scripts/sha256.html (Nov. 4, 2022)

"RSA Algorithm in Cryptography," GeeksforGeeks. https://www.geeksforgeeks.org/rsa-algorithm-cryptography/ (Apr. 18, 2022)