SECURITY AND PRIVACY

Assignment #1 - Performance Benchmarking of Cryptography Mechanisms

Marta Sofia Vieira Longo - 202207985

Sara Catarina Vieira Táboas - 202205101

In this assigment we will measure the time AES, RSA and SHA take to process files of different sizes, using a python implementation of the encryption/decryption and hash mechanisms.

DESCRIPTION OF THE EXPERIMENTAL SETUP

- OS: macOS 14.2 23C64 arm64
- Host: Mac15,3
- Kernel: 23.2.0
- Packages: 8 (brew)
- Shell: zsh 5.9
- CPU: Apple M3
- GPU: Apple M3
- Memory: 2376MiB / 16384MiB

LIBRARIES USED IN THIS PROJECT:

In [27]:
```
import os
from os import urandom
from cryptography .hazmat. primitives .ciphers import Cipher , algorithms
from Crypto.Util.Padding import pad, unpad
import timeit
from Crypto.Cipher import AES
from cryptography.hazmat.primitives.asymmetric import rsa
import matplotlib.pyplot as plt
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
import hashlib
```

A. Generate random text files with the following sizes:

- For AES (in bytes): 8, 64, 512, 4096, 32768, 262144, 2097152
- For SHA (in bytes): 8, 64, 512, 4096, 32768, 262144, 2097152
- For RSA (in bytes): 2, 4, 8, 16, 32, 64, 128

In [28]:
```
# Generates random binary text files with size = size
def generate_random_text_file(file_path, size):
    with open(file_path, 'wb') as file:
        file.write(os.urandom(size))

# Uses the generate_random_text_files() defined above to generate 7 text
def generate_files_aes():
```

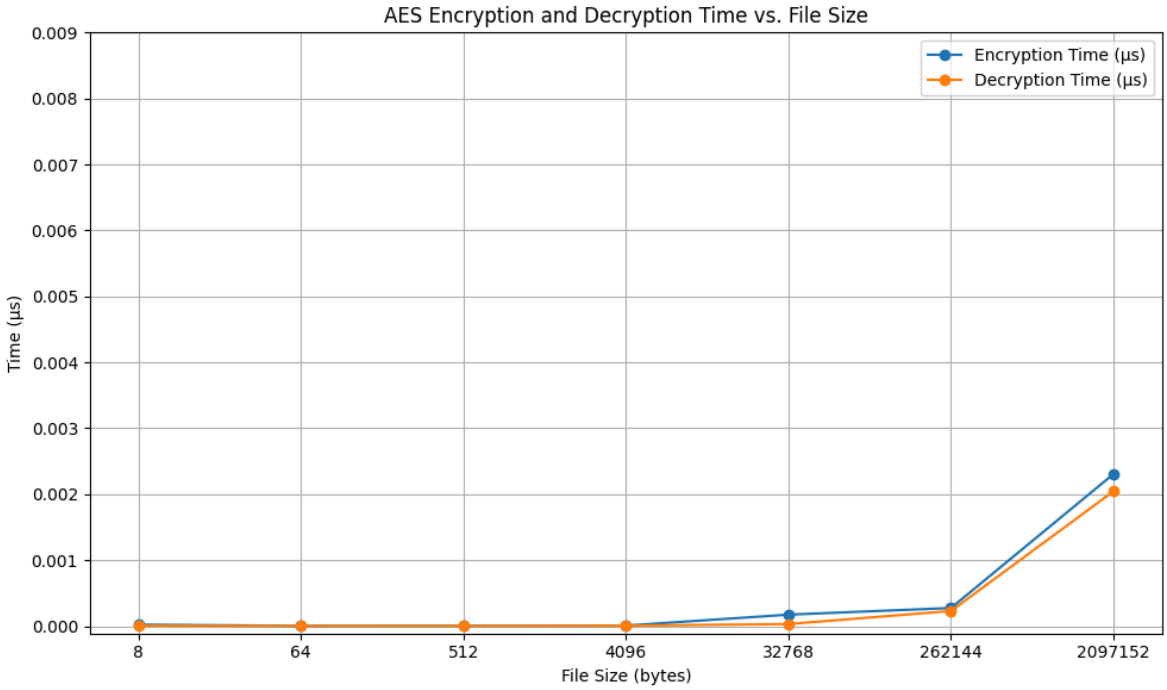
```
    aes_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]
    aes_files = []
    for size in aes_sizes:
        generate_random_text_file(f"aes_{size}.txt", size)
        aes_files.append(f"aes_{size}.txt")
    return aes_files

# Uses the generate_random_text_files() defined above to generate 7 text
def generate_files_sha():
    sha_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]
    sha_files = []
    for size in sha_sizes:
        generate_random_text_file(f"sha_{size}.txt", size)
        sha_files.append(f"sha_{size}.txt")
    return sha_files

# Uses the generate_random_text_files() defined above to generate 7 text
def generate_files_rsa():
    rsa_sizes = [2, 4, 8, 16, 32, 64, 128]
    rsa_files=[]
    for size in rsa_sizes:
        generate_random_text_file(f"rsa_{size}.txt", size)
        rsa_files.append(f"rsa_{size}.txt")
    return rsa_files
```

ENCRYPTION AND DECRYPTION USING AES

B1. Encrypt and decrypt all these files using AES. Employ a key of 256 bits. Measure the time it takes to encrypt and decrypt each of the files. To do this, you might want to use the python module timeit.

In [29]:
```
key = os.urandom(32)  # 256-bit key
iv = os.urandom(16)  # 16-byte initialization vector


# Performs AES encryption using the CFB mode and calculates the time the
def encrypt_aes(file_path, output_file, key, iv):
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend = backend
    encryptor = cipher.encryptor()
    with open(file_path, 'rb') as f_in, open (output_file, 'wb') as f_out
        data = f_in.read()
        start_time = timeit.default_timer()
        encrypted_data = encryptor.update(data) + encryptor.finalize()
        time_encrypt = timeit.default_timer() - start_time
        f_out.write(encrypted_data)
    return time_encrypt

# Performs AES decryption using CFB mode and calculates the time the algo
def decrypt_aes(input_file,outout_file, key, iv):
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=backend)
    decryptor = cipher.decryptor()
    with open (input_file, 'rb') as f_in, open(outout_file, 'wb') as f_ou
        encrypted_data = f_in.read()
        start_time = timeit.default_timer()
        decrypted_data = decryptor.update(encrypted_data) + decryptor.fin
        time_decrypt = timeit.default_timer() - start_time
```

```
    return time_decrypt


aes_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]  # File sizes in b
aes_files = generate_files_aes() # Uses the generate_file_aes() function
encryption_times_aes = []
decryption_times_aes = []
i = 0 # Control variable to access the sizes in aes_sizes

for file in aes_files: # Repeats the process to all the files in aes_file

    AES_encrypt_time = encrypt_aes(file, f'encrypted_{aes_sizes[i]}.txt',
    AES_decrypt_time = decrypt_aes(f'encrypted_{aes_sizes[i]}.txt', f'dec

    encryption_times_aes.append(AES_encrypt_time)
    decryption_times_aes.append(AES_decrypt_time)
    i+=1

# Plot of AES encryption / decryption time Vs File size
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(aes_sizes) + 1), encryption_times_aes, marker='o',
plt.plot(range(1, len(aes_sizes) + 1), decryption_times_aes, marker='o',

plt.title('AES Encryption and Decryption Time vs. File Size')
plt.xlabel('File Size (bytes)')
plt.ylabel('Time (µs)')
plt.xticks(range(1, len(aes_sizes) + 1), aes_sizes)
plt.yticks([i/1000 for i in range(10)])
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



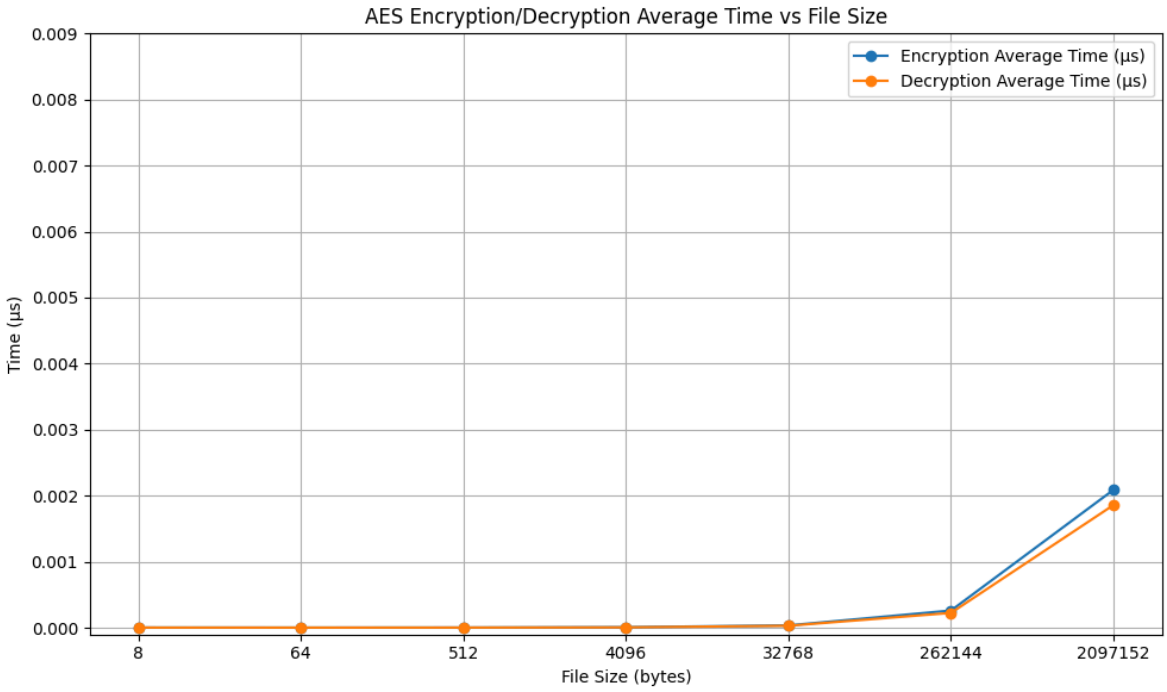
Observations:

The plot shows the time spent running the AES algorithm in CFB mode for files with different sizes.

The AES algorithm encrypts the message by dividing the plaintext into blocks and uses the key and the initialization vector to encrypt the first plaintext block, creating the first ciphertext block. Then, it uses that first ciphertext block to encrypt the second plaintext block, and it repeates this process until the end of encryption. To the decryption method (inverse process), it's not necessary to wait for the previous block, because the algorithm receives all of ciphertext blocks at once. This explains the light difference between encryption and decryption times. In addition, as the file size increases, the time AES takes also increases.

B2. Running an algorithm over the same file multiple times

In [30]:
```
key = os.urandom(32)  # 256-bit key
iv = os.urandom(16)  # 16-byte initialization vector

aes_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]  # File sizes in b
aes_files = generate_files_aes() # Uses the generate_files_aes() function

AES_encrypt_average = []
AES_decrypt_average = []
iterations = 200 # Number of times the program will perform the AES encry
i = 0 # Control variable to access the sizes in aes_sizes

for file in aes_files: # Repeats the process for each file of aes_files

    AES_encrypt_sum = 0 # Sum of the encryption times for the 200 repetit
    AES_decrypt_sum = 0 # Sum of the decryption times for the 200 repetit

    for j in range(iterations):
        AES_encrypt_sum += encrypt_aes(file, f'encrypted_{aes_sizes[i]}.t
        AES_decrypt_sum += decrypt_aes(f'encrypted_{aes_sizes[i]}.txt', f

    AES_encrypt_avg = AES_encrypt_sum / iterations # Calculates the avera
    AES_encrypt_average.append(AES_encrypt_avg)
    AES_decrypt_avg = AES_decrypt_sum / iterations # Calculates the avera
    AES_decrypt_average.append(AES_decrypt_avg)
    i+=1

# Plot of AES encryption / decryption average time Vs File size
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(aes_sizes) + 1), AES_encrypt_average, marker='o', l
plt.plot(range(1, len(aes_sizes) + 1), AES_decrypt_average, marker='o', l
plt.title('AES Encryption/Decryption Average Time vs File Size')
plt.xlabel('File Size (bytes)')
plt.ylabel('Time (µs)')
plt.xticks(range(1, len(aes_sizes) + 1), aes_sizes)
plt.yticks([i/1000 for i in range(10)])
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



Observations:

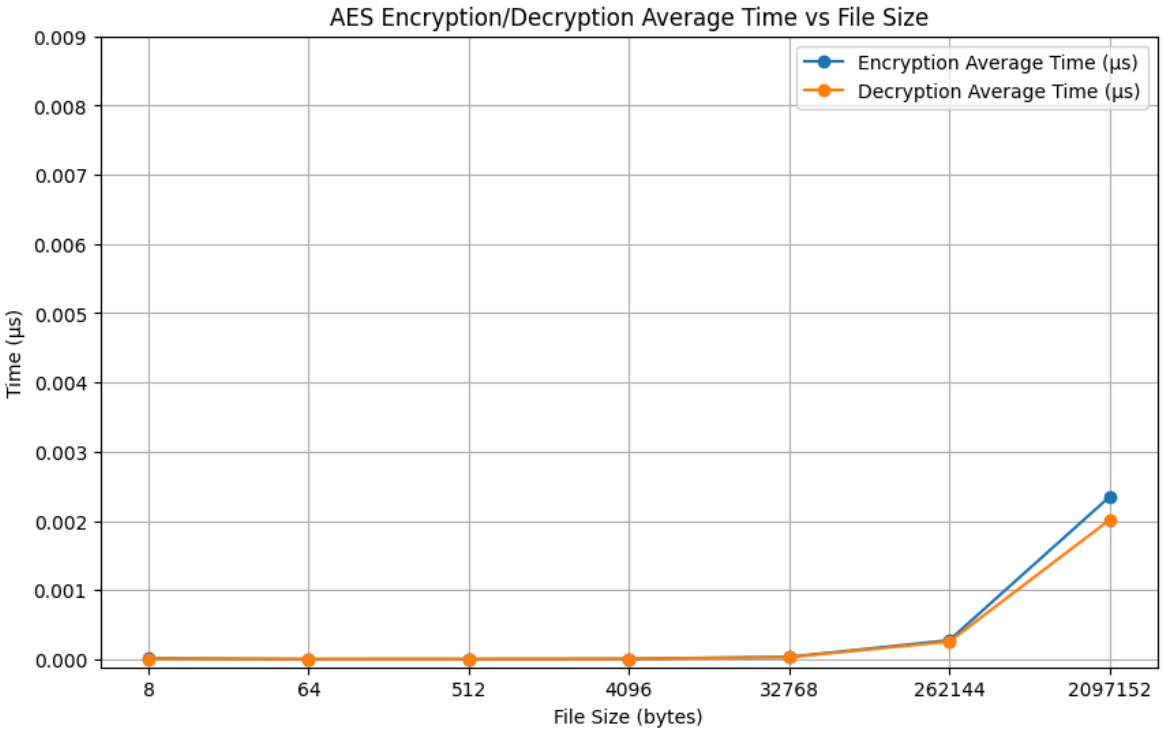
The plot shows the average times spent running the AES algorithm in CFB mode for the same file multiple times.

The average time the AES algortihm uses to encrypt / decrypt multiple times the same file of a specified size does not appear to diverge from the time it takes to encrypt / decrypt just one file - the AES behaviour seems to be the same from the one presented in section **B1**. Therefore, we can conclude that the AES algorithm doesn't change behaviour with the increase of the number of iterations.

B3. Running an algorithm over multiple randomly generated files of fixed size

In [31]:
```
aes_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]
AES_encryption_multi = []
AES_decryption_multi = []

for size in aes_sizes: # Repeats the process for each size of aes_sizes
    encryption_times = []
    decryption_times = []
    for i in range(5): # Creates 5 files of each size
        in_filename = f'{size}_multi.txt'
        generate_random_text_file(in_filename, size) # Uses the generate_
        AES_encrypt_time = encrypt_aes(in_filename, f'encrypted_{size}.tx
        AES_decrypt_time = decrypt_aes(f'encrypted_{size}.txt', f'decrypt

        # Updates the encryption/decryption times list with the times of
        encryption_times.append(AES_encrypt_time)
        decryption_times.append(AES_decrypt_time)

    # Calculate the average encryption/decryption average time for the 5
    encryption_average_time = sum(encryption_times) / len(encryption_time
    decryption_average_time = sum(decryption_times) / len(decryption_time

    # Append the averages to the overall list AES_encryption_multi and AE
```

```
    AES_encryption_multi.append(encryption_average_time)
    AES_decryption_multi.append(decryption_average_time)


# Plot of AES encryption / decryption average time VS File size
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(aes_sizes) + 1), AES_encryption_multi, marker='o',
plt.plot(range(1, len(aes_sizes) + 1), AES_decryption_multi, marker='o',
plt.title('AES Encryption/Decryption Average Time vs File Size')
plt.xlabel('File Size (bytes)')
plt.ylabel('Time (µs)')
plt.xticks(range(1, len(aes_sizes) + 1), aes_sizes)
plt.yticks([i/1000 for i in range(10)])
plt.grid(True)
plt.legend()
plt.show()
```



Observations:

The plot shows the average time spent running AES algorithm over 5 different files for each size.

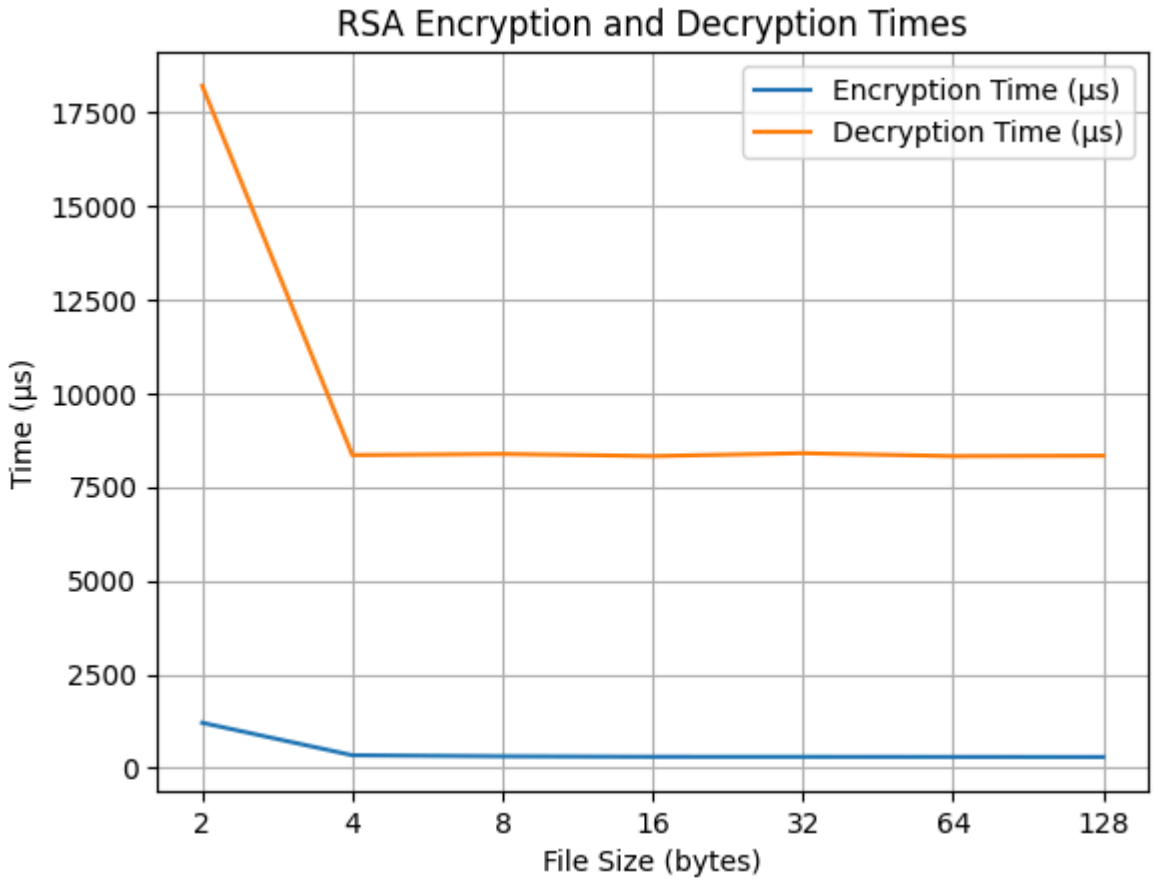
The average time the AES algortihm uses to encrypt / decrypt 5 different files of a specified size does not appear to diverge from the time it takes to encrypt / decrypt just one file - the AES behaviour seems to be the same from the one presented in section **B1**. Therefore, we can conclude that the AES algorithm is not affected for the files content.

ENCRYPTION AND DECRYPTION USING RSA

C. Using the python module for RSA encryption and decryption, measure the time of RSA encryption and decryption for the file sizes listed in part A, with a key of size 2048 bits (minimum recommended for RSA).

In [32]:
```
# Function to generate RSA key pair
def generate_rsa_key():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
    return private_key


# Function to encrypt the plaintext using the public key
def encrypt_rsa(plaintext, public_key):
    ciphertext = public_key.encrypt(
        plaintext,
        padding.OAEP( # The function uses the padding scheme OAEP (Optima
            mgf=padding.MGF1(algorithm=hashes.SHA256()), # Uses for paddi
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return ciphertext


# Function to decrypt the plaintext using the private key
def decrypt_rsa(ciphertext, private_key):
    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return plaintext



rsa_sizes = [2, 4, 8, 16, 32, 64, 128] # File sizes in bytes
encryption_times_rsa = []
decryption_times_rsa = []
rsa_files = generate_files_rsa() # Uses the generate_files_rsa() function
private_key = generate_rsa_key() # Generate the private key with generate
public_key = private_key.public_key()

# Encrypt and decrypt each file, measure the time taken
for file_path, size in zip(rsa_files, rsa_sizes):
    with open(file_path, 'rb') as f:
        file_data = f.read()

    # Measure encryption time
    start_encrypt = timeit.default_timer()
    ciphertext = encrypt_rsa(file_data, public_key)
    end_encrypt = timeit.default_timer()
    encrypt_time = (end_encrypt - start_encrypt) * 1e7 # Converting to na
    encryption_times_rsa.append(encrypt_time)

    # Measure decryption time
    start_decrypt = timeit.default_timer()
    decrypted_data = decrypt_rsa(ciphertext, private_key)
```

```
    end_decrypt = timeit.default_timer()
    decrypt_time = (end_decrypt - start_decrypt) * 1e7# Converting to nan
    decryption_times_rsa.append(decrypt_time)

# Combined plot of RSA encryption and decryption times
plt.plot(range(1, len(rsa_sizes) + 1), encryption_times_rsa, label='Encry
plt.plot(range(1, len(rsa_sizes) + 1), decryption_times_rsa, label='Decry
plt.xlabel('File Size (bytes)')
plt.ylabel('Time (µs)')
plt.xticks(range(1, len(rsa_sizes) + 1), rsa_sizes)
plt.title('RSA Encryption and Decryption Times')
plt.legend()
plt.grid(True)
plt.show()
```



Observations:

On the lowest file sizes, we can see a considerable spike because it includes the time the program uses to process the data first. The CPU takes time to process the files and the time used to access the memory is also relevant to explain this results.

In the plot, there is a significant difference between the encryption and the decryption times (explained below). In contrast, it is possible to state that the file size doesn't affect the RSA behaviour - in both processes (encryption and decryption).

Comparison between RSA encryption and decryption times:

The RSA algorithm is a assymetric encryption algorithm that relies on exponentiation and modular arithmetic and uses public /private key pair. When encrypting a message, the sender uses the public key of the recipient to compute the ciphertext.

When decrypting an RSA encrypted message, the recipient uses their private key to compute the plaintext message.

The time differences between encryption and decryption in the RSA algorithm primarily arise from the differences in exponentiation operations, key sizes, and modular arithmetic computations.

1. **Encryption:** During RSA encryption, the computation involves raising the plaintext message to the power of the public exponent. This operation is relatively efficient, especially if the public exponent is chosen to be a small value like 65537. Therefore, encryption tends to be faster due to the use of smaller exponents and simpler arithmetic operations.

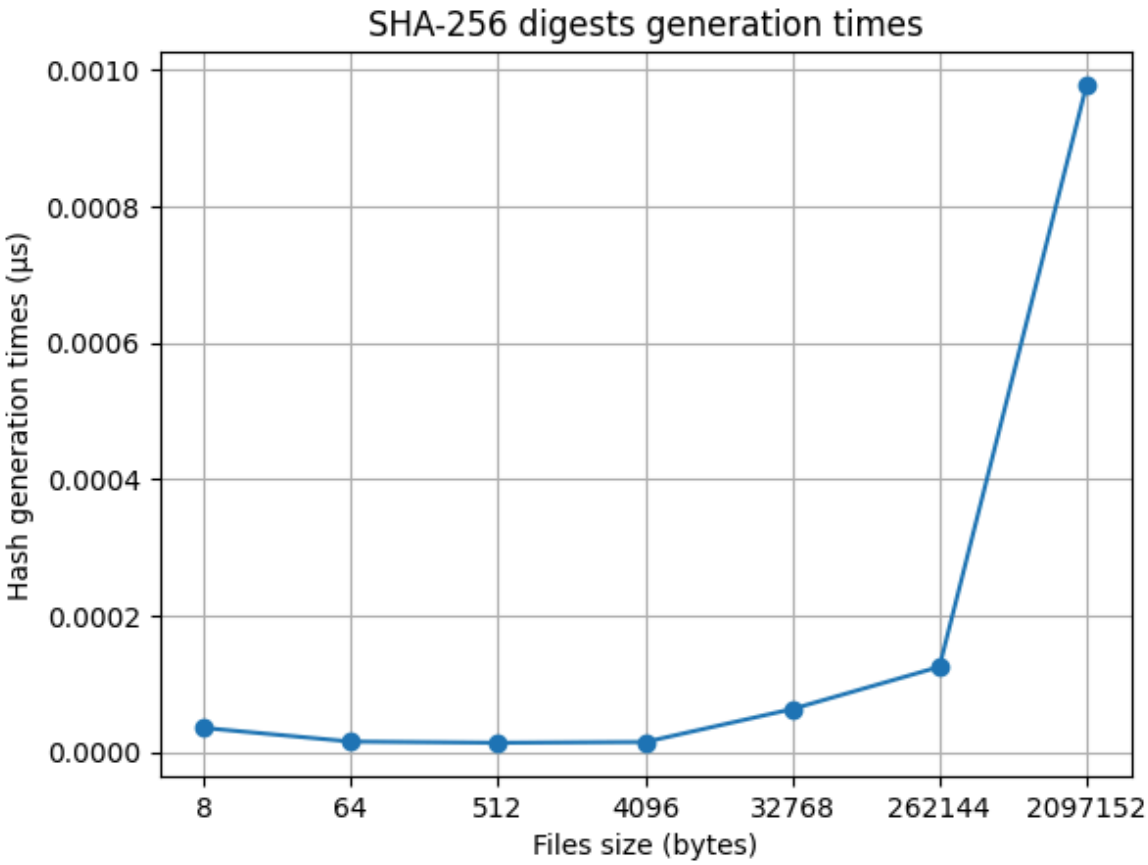
2. **Decryption:** During RSA decryption, the computation involves raising the ciphertext to the power of the private exponent. The private exponent tends to be significantly larger than the public exponent, resulting in more computationally intensive exponentiation operations. Therefore, decryption requires handling larger exponents and more complex arithmetic operations, resulting in longer computation times - explaining the times differences in the plot above.

ENCRYPTION AND DECRYPTION USING SHA

D. Measure the time for SHA-256 hash generation for the file sizes listed in part A.

In [33]:
```
# Function to calculate the SHA-256 hash and convert it to a hexadecimal
def calculate_sha256_hash(file_path):
    with open(file_path, 'rb') as f:
        data = f.read()
    sha256_hash = hashlib.sha256(data).hexdigest()
    return sha256_hash

# Function to calculate SHA-256 hashes for a list of files and measure th
def calculate_sha256_hash_file_list(file_list):
    sha_hash_times = []
    for file in file_list:
        start_time = timeit.default_timer()
        calculate_sha256_hash(file)
        end_time = timeit.default_timer()
        sha_hash_times.append(end_time - start_time)
    return sha_hash_times

sha_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152] # File sizes in by
sha_files = generate_files_sha() # Uses the generate_files_sha() defined
sha_hash_times = calculate_sha256_hash_file_list(sha_files)

plt.plot(range(1, len(sha_sizes) + 1), sha_hash_times, marker='o')
plt.xlabel('Files size (bytes)')
plt.ylabel('Hash generation times (µs)')
plt.xticks(range(1, len(sha_sizes) + 1), sha_sizes)
plt.title('SHA-256 digests generation times')
```

```
plt.grid(True)
plt.show()
```



Observations:

SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic hash function that produces a 256-bit (32-byte) hash value from input data. SHA-256 operates by processing input data in blocks of 512 bits and producing a 256-bit hash value as output. It is designed to be computationally secure, meaning it is computationally infeasible to generate two different inputs that produce the same hash value (collision resistance).

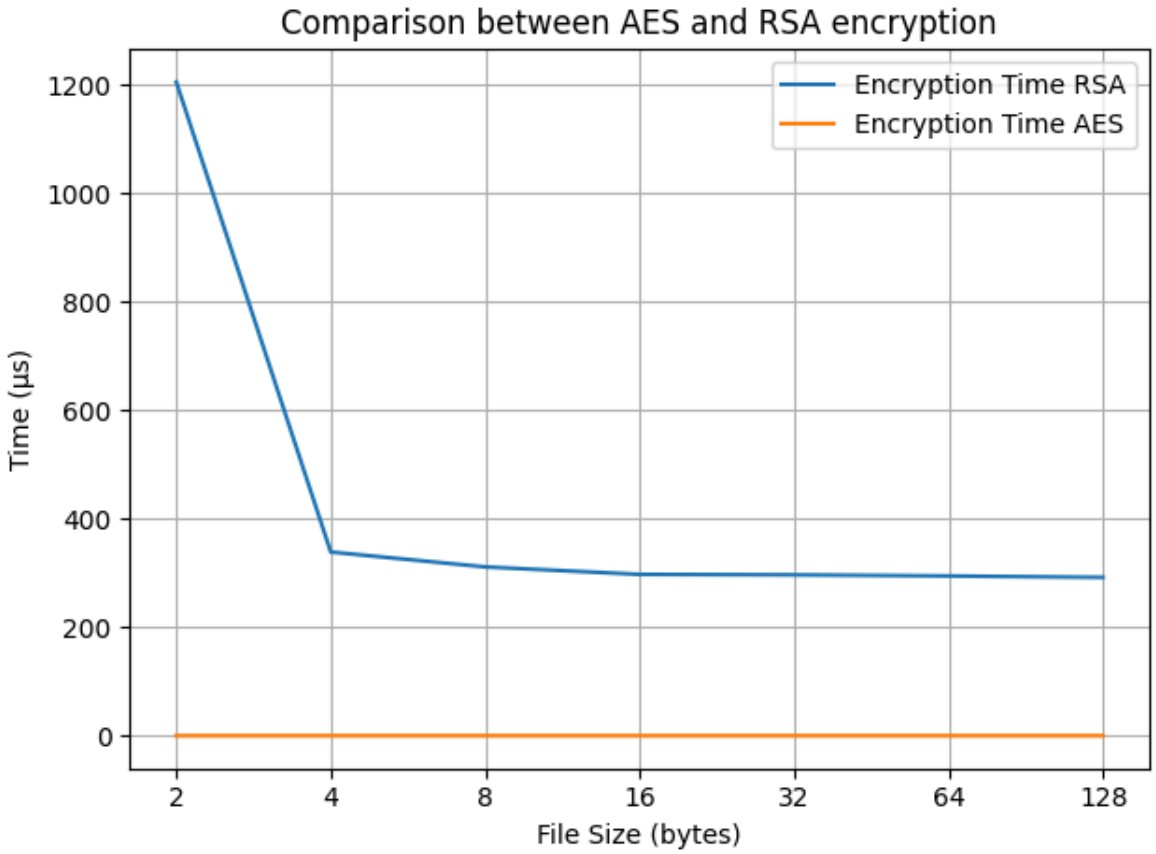
The plot shows the hash generation times to files with different sizes. On the first file size, we can see a light spike because it includes the time the program uses to process the data first. The CPU takes time to process the files and the time used to access the memory is also relevant to explain this results.

Through the resulting line, it's possible to state that the hash generation time tends to increase with the increase of file size.

Comparison between AES encryption and RSA encryption

In [36]:
```
plt.plot(range(1, len(rsa_sizes) + 1), encryption_times_rsa, label='Encry
plt.plot(range(1, len(aes_sizes) + 1), encryption_times_aes, label='Encry
plt.xlabel('File Size (bytes)')
plt.ylabel('Time (µs)')
plt.xticks(range(1, len(rsa_sizes) + 1),rsa_sizes)
plt.title('Comparison between AES and RSA encryption')
plt.tight_layout()
```

```
plt.legend()
plt.grid(True)
plt.show()
```



Analysis and Conclusions:

The plot shows AES and RSA encryption times through several files of different sizes. We can state that the RSA algorithm takes more time encrypting a file than AES, due to the differences in their process.

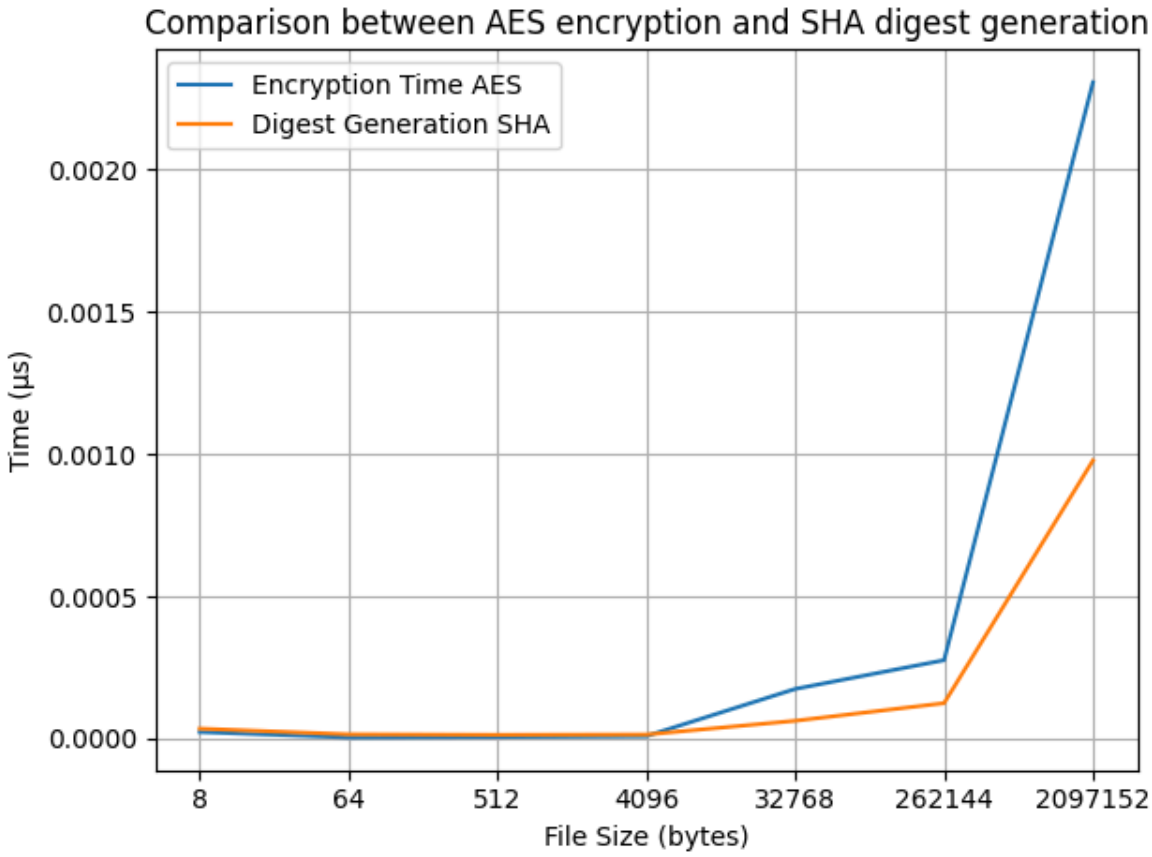
AES encryption involves simple operations, such as substitution, permutations and XOR. As a method of symmetric encryption, it's usually faster than other processes, because it requires less computational power, making it suitable for encrypting large amounts of data.

On the other hand, RSA encryption consists in an asymmetric encryption method. Therefore it takes more time, since it requires a key that has to be generated, with a higher length than the one used in AES. RSA is used to situations in which security is the priority over the encryption time.

Another reason why the AES takes less time can be the complexity of RSA when compared to the AES algothimn which in consequence makes the RSA more safer because is very difficult for someone without the private key to acess to the information.

Comparison between AES encryption and SHA digest generation

In [35]:
```
plt.plot(range(1, len(aes_sizes) + 1), encryption_times_aes, label='Encry
plt.plot(range(1, len(sha_sizes) + 1), sha_hash_times, label='Digest Gene
plt.xlabel('File Size (bytes)')
plt.ylabel('Time (µs)')
plt.xticks(range(1, len(sha_sizes) + 1),sha_sizes)
plt.title('Comparison between AES encryption and SHA digest generation')
plt.tight_layout()
plt.legend()
plt.grid(True)
plt.show()
```



Analysis and Conclusions:

The plot shows AES encryption time and SHA digest generation through several files of different sizes. We can state that AES encryption takes more time than SHA-256 digest generation, specially when talking about larger file sizes.

This happens because SHA-256 is a cryptographic hash function that generates a fixed-size hash value from input data. It involves a relatively straightforward set of operations, such as bitwise logical operations, rotations, and modular addition. On the other hand, AES encryption requires multiple rounds of processing for each block of data, with each round involving various operations, such as substitution, permutations and XOR.

FINAL CONCLUSION

In this project, we conducted a comprehensive analysis of the performance of symmetric (AES encryption) and asymmetric (RSA encryption and decryption) cryptography algorithms, along with SHA-256 hash generation, using Python's

cryptography module. We measured the time taken for cryptographic operations on files of varying sizes and key lengths to understand their efficiency and scalability.

AES Encryption and Decryption: We observed that AES encryption and decryption times remained relatively consistent across different file sizes. AES proved to be efficient and suitable for securing data, especially for large files. However, for very small files, padding overhead might affect performance.

RSA Encryption and Decryption: RSA encryption and decryption times increased significantly with larger file sizes, as expected due to the computational complexity of asymmetric cryptography. RSA encryption was notably slower compared to AES encryption, emphasizing the trade-off between security and performance in asymmetric encryption.

SHA-256 Hash Generation: SHA-256 hash generation exhibited consistent performance across different file sizes, as it operates independently of file size. It provided a fast and reliable means for generating cryptographic digests, essential for data integrity verification.