crsx30o6m

April 6, 2025

ASSIGNMENT #1: Performance Benchmarking of Cryptographic Mechanisms

Group: Carlos Ortega, David Sá, João Morais

Setup for experiment OS: Windows 11 Pro 64-bit
Host: Lenovo Legion 7i Gen 8
Kernel: Windows NT 10.0 (Build 22621)
Packages: Lenovo Vantage, Legion Toolkit
Shell: PowerShell 5.1 / Windows Terminal
CPU: Intel Core i9-13900HX (24 cores, 32 threads)
GPU: NVIDIA GeForce RTX 4080 Laptop GPU + Intel UHD Graphics
Memory: 64384MiB / 65536MiB DDR5 5600MHz
Storage: 476GiB / 500GiB PCIe Gen4 NVMe SSD
Display: 16" WQXGA (2560x1600), 240Hz, 100% sRGB
Cooling: Legion Coldfront 5.0 (vapor chamber + AI-tuning)

Introduction In this project, we are comparing the performance of three widely used cryptographic algorithms: AES (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), and SHA (Secure Hash Algorithm). We want to know how long it takes each algorithm to encrypt files of different sizes and how efficient they are.

AES is a symmetric block cipher that is secure and fast, therefore commonly used to encrypt data. RSA is an asymmetric encryption scheme used for secure key exchange and digital signatures but normally slower since it relies on math-focused operations. SHA-256 is a cryptographic hash function that offers data integrity by generating a special fingerprint for every file.

To benchmark, we generate random files of specified sizes and carry out encryption, decryption, and hashing operations. We measure execution time and see how output changes when an algorithm is run an incredibly large number of times on the same file as opposed to varying randomly generated files of the same size.

The following report includes:

AES, RSA, and SHA-256 implementation in Python Description of test setup, i.e., hardware and software requirements Performance benchmarking with graphical result presentation AES vs. RSA encryption, AES encryption vs. SHA hashing, RSA encryption vs. decryption comparison

In this research, we would like to give an insight into the effectiveness of different cryptographic processes and their applicability across different applications.

Libraries and Tools Used

[32]:
```
import os
from os import urandom
from cryptography .hazmat. primitives .ciphers import Cipher , algorithms ,␣
 ↪modes
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 FILES 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

[33]:
```
# 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 files␣
 ↪of different sizes (aes_sizes). Saves those files in an array files_aes
def generate_files_aes():
    aes_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]
    files_aes = []
    for size in aes_sizes:
# Recorremos los caracteres del texto
        generate_random_text_file(f"aes_{size}.txt", size)
        files_aes.append(f"aes_{size}.txt")
    return files_aes

# Uses the generate_random_text_files() defined above to generate 7 text files␣
 ↪of different sizes (sha_sizes). Saves those files in an array files_sha
def generate_files_sha():
    sha_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]
    files_sha = []
    for size in sha_sizes:
# Recorremos los caracteres del texto
        generate_random_text_file(f"sha_{size}.txt", size)
        files_sha.append(f"sha_{size}.txt")
```

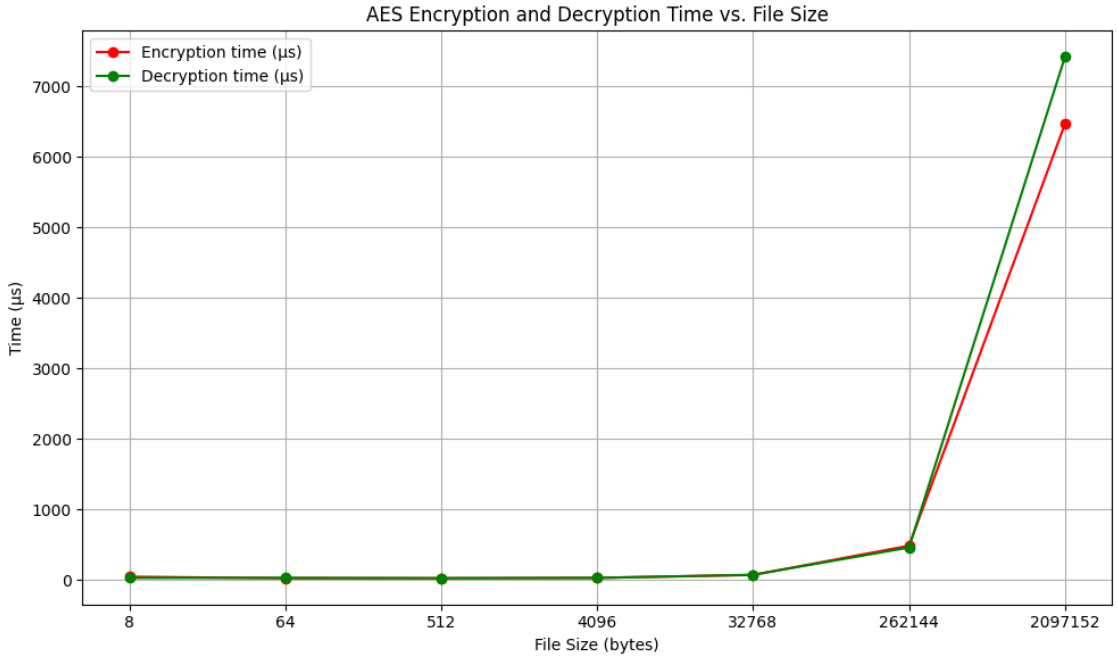
```
    return files_sha

# Uses the generate_random_text_files() and saves those files in an array␣
 ↪files_rsa
def generate_files_rsa():
    rsa_sizes = [2, 4, 8, 16, 32, 64, 128]
    files_rsa=[]
    for size in rsa_sizes:
# Recorremos los caracteres del texto
        generate_random_text_file(f"rsa_{size}.txt", size)
        files_rsa.append(f"rsa_{size}.txt")
    return files_rsa
```

B. 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.

B1. ENCRYPTION OF MULTIPLE FILES OF FIXED SIZE

[34]:
```
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␣
 ↪algorithm takes to encrypt a file
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) * 1e6
        f_out.write(encrypted_data)
    return time_encrypt

# Performs AES decryption using CFB mode
# Calculates the time the algorithm takes to decrypt a file
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_out:
        encrypted_data = f_in.read()
        start_time = timeit.default_timer()
        decrypted_data = decryptor.update(encrypted_data) + decryptor.finalize()
```

```
        time_decrypt = (timeit.default_timer() - start_time) * 1e6
    return time_decrypt


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


for file in files_AES: # Repeats the process to all the files in files_AES

    AES_encrypt_time = encrypt_aes(file, f'encrypted_{aes_sizes[i]}.txt', key,␣
 ↪iv) # Calculates the encryption time for a file
    AES_decrypt_time = decrypt_aes(f'encrypted_{aes_sizes[i]}.txt',␣
 ↪f'decrypted_{aes_sizes[i]}.txt', key, iv)

    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',␣
 ↪label='Encryption time (µs)', color='red')
plt.plot(range(1, len(aes_sizes) + 1), decryption_times_aes, marker='o',␣
 ↪label='Decryption time (µs)', color='green')
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.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



Observations: The graphic show the time spent running the AES algorithm in CBF mode, for file with different sizes.

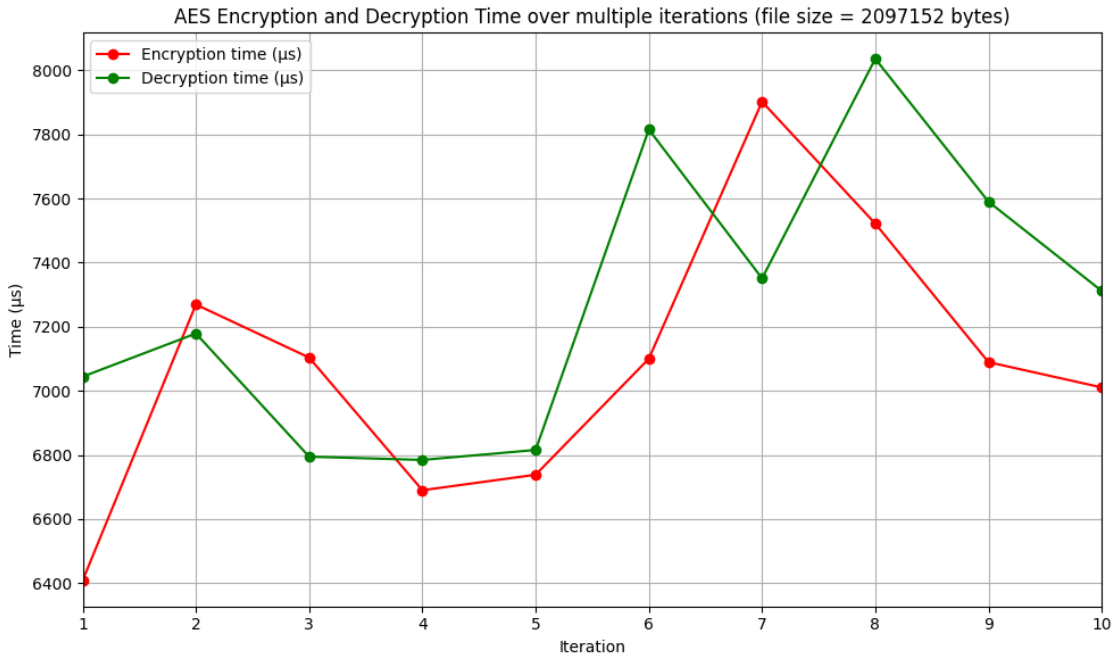
Encryption Process: The AES algorithm encrypts the message by dividing the plaintext into blocks. It uses a key and a inicialization vector to encrypt the first block, generating the first ciphertext block. This ciphertext block is then used to encrypt the next plaintext block, this process will repeat until the entire message is encrypted.

Decryption Process: Unlike the encryption process, decryption does not require waiting for the previous block, all the cyphertext blocks are processed simultaneously, this explains the light difference between encryption and decryption times.

B2. Running an algorithm over the same file multiple times

[35]:
```
import os
import timeit
import matplotlib.pyplot as plt
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

key = os.urandom(32)  # 256-bit key
iv = os.urandom(16)  # Vector de inicialización de 16 bytes

aes_size = 2097152  # Tamaños de archivo en bytes
generate_files_aes()  # Genera archivos de prueba
encryption_times_aes = []
decryption_times_aes = []
```

```
i = 0  # Control de índice para acceder a los tamaños

# Repite el proceso para todos los archivos generados
for i in range(10):
# Recorremos los caracteres del texto
    AES_encrypt_time = encrypt_aes(file, f'encrypted_{aes_size}.txt', key, iv) ␣
 ↪# Tiempo de encriptación
    AES_decrypt_time = decrypt_aes(f'encrypted_{aes_size}.txt',␣
 ↪f'decrypted_{aes_size}.txt', key, iv)  # Tiempo de desencriptación

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

# Gráfico: Tiempo de encriptación / desencriptación AES vs Tamaño del archivo
plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), encryption_times_aes, marker='o', label='Encryption time␣
 ↪(µs)', color='red')
plt.plot(range(1, 11), decryption_times_aes, marker='o', label='Decryption time␣
 ↪(µs)', color='green')
plt.title('AES Encryption and Decryption Time over multiple iterations (file␣
 ↪size = 2097152 bytes)')
plt.xlabel('Iteration')
plt.ylabel('Time (µs)')
plt.xlim(1,10)
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



Observations The graph displays the execution times (in microseconds, µs) for encryption and decryption operations performed repeatedly on the same file over 10 iterations. Both processes show a progressive reduction in execution time, suggesting potential performance gains with consecutive runs.

Key Observations:

First iteration: Encryption: ~7800 µs Decryption: ~8000 µs (assuming the top line represents decryption, as the legend is not explicit).

Tenth iteration: Both processes reach ~6400–6800 µs, showing a reduction of ~15–20% compared to the initial run.

Possible Causes for the Observed Trend:

Cache Optimizations: The system may leverage CPU cache or memory buffers after the first few executions, speeding up data access. File I/O operations (read/write) could become faster due to filesystem optimizations.

Hardware/Software Warm-up: Cryptographic libraries (e.g., OpenSSL, AES-NI) may optimize internal processes, such as key expansion, after repeated use. The processor itself might adjust its frequency (e.g., turbo boost) or resource allocation under sustained workloads.

Consistency Between Encryption and Decryption: The symmetrical reduction in time suggests both processes share similar dependencies (e.g., accessing the same file, hardware resource usage).

C. RSA ENCRYPTION AND DECRYPTION

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).

[36]:
```
import timeit
import matplotlib.pyplot as plt
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend

# 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 (Optimal␣
 ↪Asymmetric Encryption Padding), usually used in RSA to enhance security and␣
 ↪prevent certain cryptographic attacks.
            mgf=padding.MGF1(algorithm=hashes.SHA256()),  # Uses for padding␣
 ↪the hash algorithm SHA-256
            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␣
 ↪defined in A. to create a list of files
private_key = generate_rsa_key()  # Generate the private key with␣
 ↪generate_rsa_key()
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):
# Recorremos los caracteres del texto
    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) * 1e6  # Converting to␣
 ↪microseconds
    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) * 1e6  # Converting to␣
 ↪microseconds
    decryption_times_rsa.append(decrypt_time)

# Combined plot of RSA encryption and decryption times (titles and labels in␣
 ↪English)
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(rsa_sizes) + 1), encryption_times_rsa, marker='o',␣
 ↪label='Encryption time (µs)', color='red')
plt.plot(range(1, len(rsa_sizes) + 1), decryption_times_rsa, marker='o',␣
 ↪label='Decryption time (µs)', color='green')
plt.title('RSA Encryption and Decryption Time vs File Size')
plt.xlabel('File size (bytes)')
plt.ylabel('Time (µs)')
plt.xticks(range(1, len(rsa_sizes) + 1), rsa_sizes)
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Observations Execution times for smaller files show a noticeable spike due to the initial processing and memory access overhead. The graph also highlights a significant difference between encryption and decryption times. However, RSA performance remains unaffected by file size, as both encryption and decryption times stay consistent regardless of the input size.

Comparison between RSA encryption and decryption times The RSA algorithm is an asymmetric encryption method that relies on modular arithmetic and exponentiation, utilizing a pair of public and private keys. To encrypt a message, the sender applies the recipient's public key to transform the plaintext into ciphertext. To decrypt the message, the recipient uses their private key to recover the original plaintext.

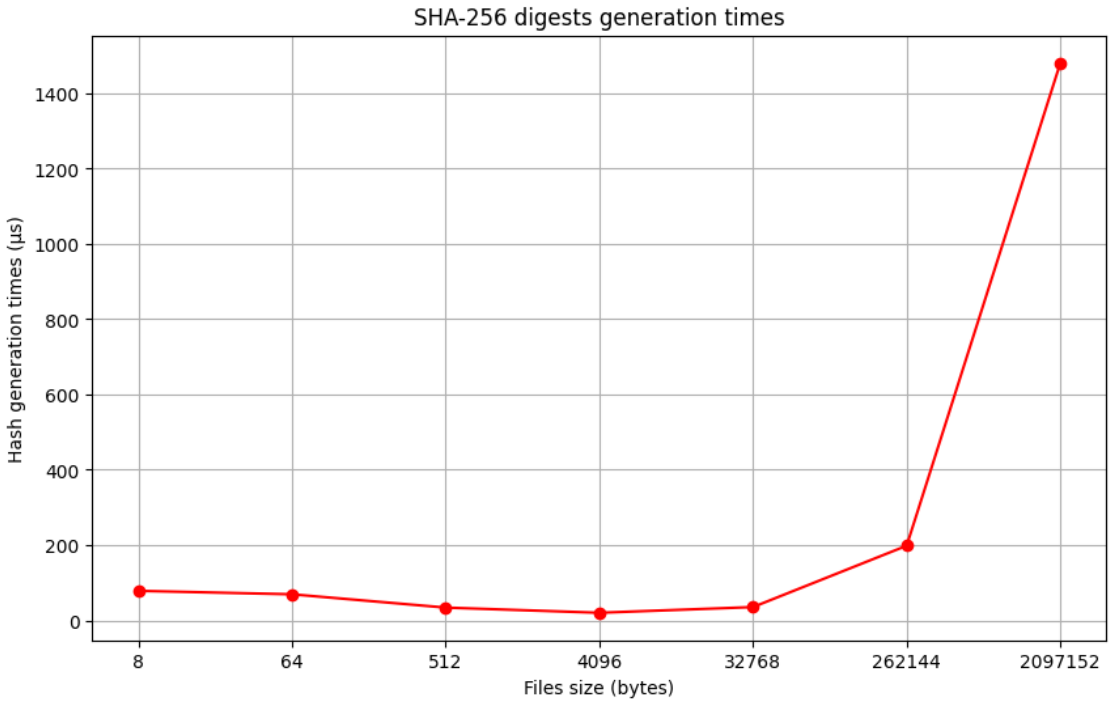
The variation in encryption and decryption times in RSA stems mainly from differences in exponentiation operations, key sizes, and modular arithmetic complexity.

Encryption: The encryption process involves raising the plaintext to the power of the public exponent. Since the public exponent is often a small value, such as 65537, the computation is relatively efficient. This results in encryption being generally faster due to the simpler arithmetic involved.

Decryption: In contrast, decryption requires raising the ciphertext to the power of the private exponent. The private exponent is usually much larger than the public one, making the exponentiation process significantly more computationally demanding. As a result, decryption takes longer, explaining the time differences observed in the graph.

D. SHA-256 time for hash generation

[37]:
```
# Function to calculate the SHA-256 hash and convert it to a hexadecimal string
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 the time␣
 ↪taken
def calculate_sha256_hash_file_list(file_list):
    sha_hash_times = []
    for file in file_list:
# Recorremos los caracteres del texto
        start_time = timeit.default_timer()
        calculate_sha256_hash(file)
        end_time = timeit.default_timer()
        sha_hash_times.append((end_time - start_time) * 1e6)
    return sha_hash_times

sha_sizes = [8, 64, 512, 4096, 32768, 262144, 2097152] # File sizes in bytes
sha_files = generate_files_sha() # Uses the generate_files_sha() defined in A.␣
 ↪to create a list of files
sha_hash_times = calculate_sha256_hash_file_list(sha_files)

plt.figure(figsize=(10, 6))
plt.plot(range(1, len(sha_sizes) + 1), sha_hash_times, marker='o', color= 'red')
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()
```



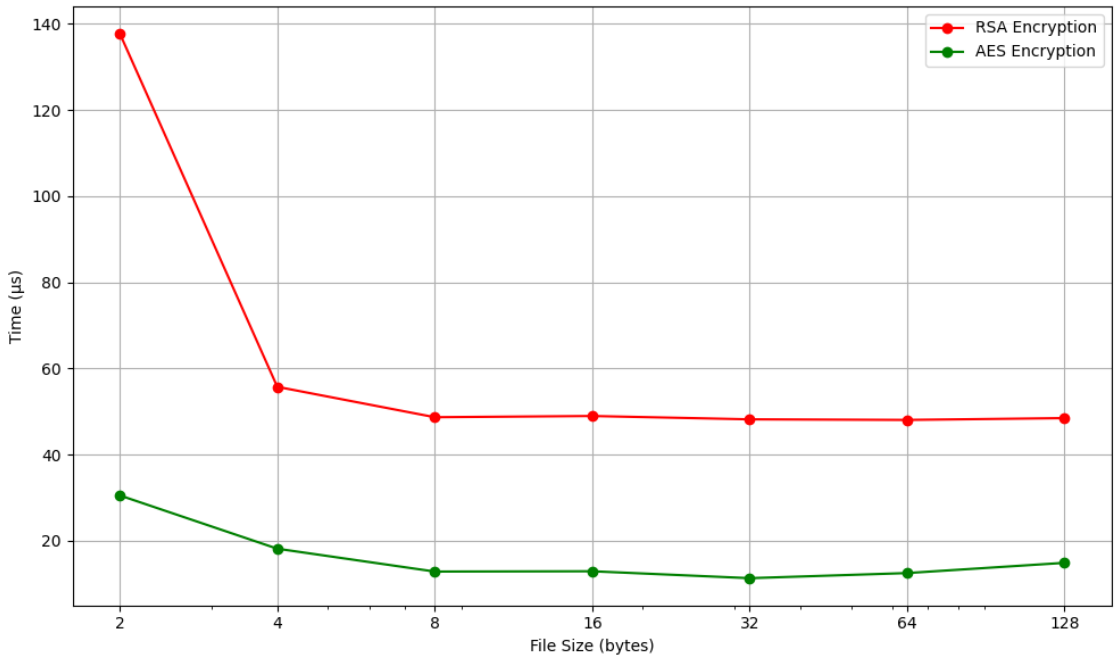
Observations The image shows the hash generation times to files with different sizes. For the smallest file size, there is a slight spike in execution time due to the initial processing overhead. The CPU requires time to handle the data, and memory access latency also plays a role in this result.

From the observed trend, it is evident that the time required for hash generation generally increases as the file size grows.

Comparison between AES encryption and RSA encryption Here we decided to create additional files for AES to match RSA's own file sizes, in order to make a fair comparison

[38]:
```
aes_sizes_custom = [2, 4, 8, 16, 32, 64, 128]  # File sizes in bytes
def generate_files_aes_custom():
    files_aes = []
    for size in aes_sizes_custom:
    # Recorremos los caracteres del texto
        generate_random_text_file(f"aes_{size}.txt", size)
        files_aes.append(f"aes_{size}.txt")
    return files_aes
files_AES = generate_files_aes_custom()
encryption_times_aes_custom = []
decryption_times_aes_custom = []
i = 0
for file in files_AES:
```

```
    AES_encrypt_time = encrypt_aes(file, f'encrypted_{aes_sizes_custom[i]}.
 ↪txt', key, iv)
    AES_decrypt_time = decrypt_aes(f'encrypted_{aes_sizes_custom[i]}.txt',␣
 ↪f'decrypted_{aes_sizes_custom[i]}.txt', key, iv)

    encryption_times_aes_custom.append(AES_encrypt_time )
    decryption_times_aes_custom.append(AES_decrypt_time)
    i+=1

plt.figure(figsize=(10, 6))
plt.plot(rsa_sizes, encryption_times_rsa, 'o-', label='RSA Encryption',␣
 ↪color='red')
plt.plot(aes_sizes_custom, encryption_times_aes_custom, 'o-', label='AES␣
 ↪Encryption',color='green')
plt.xscale('log')
plt.xlabel('File Size (bytes)')
plt.ylabel('Time (µs)')
plt.xticks(aes_sizes_custom, aes_sizes_custom)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



ANALYSIS AND CONCLUSIONS

The plot compares AES and RSA encryption times across various file sizes, showing that RSA takes significantly longer than AES due to fundamental differences in their encryption processes.

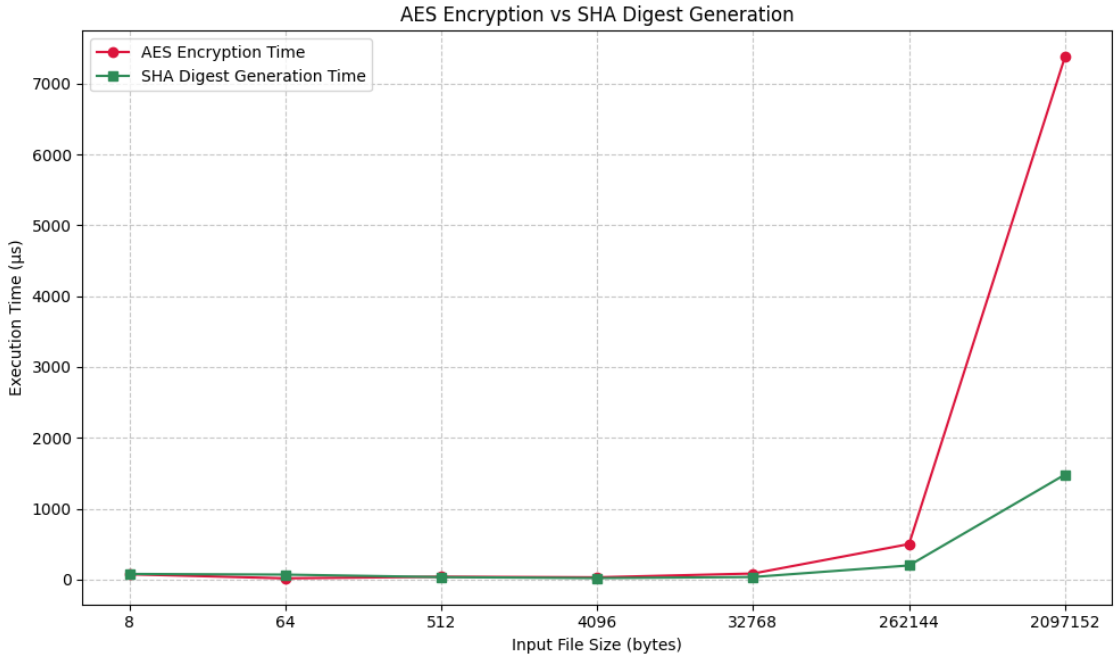
AES, a symmetric encryption algorithm, relies on efficient operations like substitution, permutation, and XOR. Because it uses the same key for encryption and decryption, it requires less computational power, making it well-suited for encrypting large datasets quickly.

In contrast, RSA is an asymmetric encryption method, meaning it requires key pair generation and uses larger key sizes than AES. This additional complexity makes RSA slower but enhances security, making it ideal for situations where protecting data is more important than encryption speed.

Another factor contributing to AES's efficiency is that RSA's computational complexity is significantly higher. However, this complexity also makes RSA more secure, as decrypting the data without the private key is extremely difficult.

Comparison between AES encryption and SHA digest generation

[39]:
```
files_AES = generate_files_aes()
encryption_times_aes = []
decryption_times_aes = []
i = 0 # Control variable to access the sizes in aes_sizes


for file in files_AES: # Repeats the process to all the files in files_AES

    AES_encrypt_time = encrypt_aes(file, f'encrypted_{aes_sizes[i]}.txt', key,␣
 ↪iv) # Calculates the encryption time for a file
    AES_decrypt_time = decrypt_aes(f'encrypted_{aes_sizes[i]}.txt',␣
 ↪f'decrypted_{aes_sizes[i]}.txt', key, iv)

    encryption_times_aes.append(AES_encrypt_time )
    decryption_times_aes.append(AES_decrypt_time)
    i+=1
plt.figure(figsize=(10, 6))  # Define o tamanho da figura para consistência␣
 ↪visual
plt.plot(range(1, len(aes_sizes) + 1), encryption_times_aes, marker='o',␣
 ↪label='AES Encryption Time', color='crimson')
plt.plot(range(1, len(sha_sizes) + 1), sha_hash_times, marker='s', label='SHA␣
 ↪Digest Generation Time', color='seagreen')
plt.xlabel('Input File Size (bytes)')
plt.ylabel('Execution Time (µs)')
plt.xticks(range(1, len(sha_sizes) + 1), sha_sizes)
plt.title('AES Encryption vs SHA Digest Generation')
plt.legend(loc='best')
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



ANALYSIS AND CONCLUSIONS

The plot compares AES encryption time and SHA-256 digest generation across different file sizes, showing that AES encryption takes longer, especially for larger files.

SHA-256, a cryptographic hash function, produces a fixed-size hash from input data using relatively simple operations like bitwise logic, rotations, and modular addition. In contrast, AES encryption processes data in multiple rounds, applying complex transformations such as substitution, permutation, and XOR at each step. This additional processing makes AES slower compared to SHA-256.

CONCLUSION In this study, we took a close look at the performance of three key cryptographic operations: AES encryption and decryption, RSA encryption and decryption, and SHA-256 hash generation. Each of these methods is essential for keeping information secure, and they each have their own unique traits when it comes to efficiency and computational complexity.

AES, which is a symmetric encryption algorithm, really shines in terms of efficiency, especially when you stack it up against RSA. Its design, which involves substitutions, permutations, and XOR operations, allows for quick encryption and decryption of large amounts of data, making it the go-to choice for applications where speed is essential.

On the flip side, RSA, being an asymmetric encryption method, tends to have much longer execution times. This is largely due to the intricate mathematical operations involved, like modular exponentiation with lengthy keys. However, this complexity is worth it in situations where security is paramount, such as in secure key exchanges and data authentication.

SHA-256 really impressed us with its speed in generating hashes. As a cryptographic hash function, it actually performed faster than AES, particularly with larger files. Its biggest strength lies in its

ability to ensure data security and integrity, since even the tiniest change in the input results in completely different hash values.

To wrap it up, each of the techniques we examined has its own specific role in the world of cryptography. While AES strikes a balance between security and performance for encrypting large-scale data, RSA provides robust protection for sensitive information, and SHA-256 guarantees data integrity. The best choice really depends on the context and the particular needs of the application.