Performance Benchmarking of Cryptographic Mechanisms

Trabalho realizado pelos alunos Maximiliano Vítor Phillips e Sá (up202305979),Rita Maria Pinho Moreira (up202303885) e Samuel José Sousa Ventura da Silva (up202305647).

0. Índice

1. Introdução

Este projeto tem como objetivo analisar o desempenho dos algoritmos AES, RSA e SHA-256 na encriptação/desencriptação e hashing de arquivos de diferentes tamanhos. Para avaliar o desempenho dos algoritmos AES, RSA e SHA-256, desenvolvemos funções em Python utilizando as bibliotecas **cryptography, hashlib, matplotlib, numpy e timeit**. Medimos o tempo de execução dos processos de encriptação, desencriptação e hashing em ficheiros de diferentes tamanhos. Os resultados foram visualizados por meio de gráficos gerados com **matplotlib** para facilitar a interpretação dos dados.

AES (Advanced Encryption Standard)

O AES é um algoritmo de criptografia simétrica, ou seja, usa a mesma chave para encriptação e decriptação. Ele opera com blocos de 128 bits e suporta chaves de 128, 192 ou 256 bits. O AES é amplamente utilizado devido à sua segurança, velocidade e eficiência em hardware e software. Um dos modos mais utilizados neste algoritmo é o CBC (Cipher Block Chaining). Este divide os dados a serem criptografados em blocos de 16 bytes, sendo o primeiro dos blocos combinado com um vetor de inicialização, via XOR, antes de ser criptografado Isso permite um encadeamento de blocos, em que cada bloco de texto cifrado é usado para modificar o próximo bloco antes da criptografia.

RSA (Rivest-Shamir-Adleman)

O RSA é um algoritmo de criptografia assimétrica, ou seja, usa um par de chaves: uma chave pública para encriptação e uma chave privada para decriptação. O RSA suporta chaves de 2048, 3072 e 4096 bits mas uma chave de 2048 bits é considerada suficientemente segura e é o tamanho mais utilizado. É uma das mais antigas e mais utilizadas formas de transmisão segura de dados. O sistema de encriptação utiliza uma factorização de dois números primos e não há nenhum método publicado para hackear o sistema devido à chave grande.

SHA (Secure Hash Algorithm)

O SHA é uma família de funções de hash criptográfico desenvolvida pela NSA (National Security Agency). A sua principal função é converter dados de qualquer tamanho num tamanho fixo de bits (hash), garantindo integridade e autenticidade. Este algoritmo possui 3 grupos: SHA-1, de 160 bits, considerado por muitos inseguro; SHA-2, que vai de 224 bits a 512 bits, bastante seguro e usado; e SHA-3, com os mesmos bits do grupo anterior, mas mais resistente a ataques.

O trabalho foca-se no SHA-256, que gera um hash de 256 bits. É amplamente utilizado em assinaturas digitais, blockchain e verificação de integridade de arquivos. Atualmente, é considerado muito seguro, sem ataques práticos conhecidos.

---

2. Implementação

Geração de Arquivos

Foram gerados, para este projeto, 100 ficheiros para cada tamanho pedido, através do bloco de código abaixo. Este cria, para cada tamanho constatado no array file_sizes, uma pasta para o mesmo, onde gera e armazena os 100 ficheiros.

In [ ]:
```
import os
import random
import string

def generate_random_file(folder, filename, size):
    file_path = os.path.join("text_files", folder, filename)
    os.makedirs(os.path.dirname(file_path), exist_ok=True)

    random_content = ''.join(
        random.choices(string.ascii_letters + string.digits,
                       k=size)
```

```
        )
    with open(file_path, "w") as f:
        f.write(random_content)

file_sizes = [2, 4, 8, 16, 32, 64, 128, 512,
              4096, 32768, 262144, 2097152]
for size in file_sizes:
    folder_name = str(size)  # Folder named after size
    for i in range(1, 101):
        generate_random_file(folder_name,
                             f"{size}_{i}.txt", size)
```

Encriptação e Desencriptação com AES

Para a implementação das funções de AES, são necessárias bibliotecas e módulos, tal como algumas variáveis globais, que permitem o acesso às mesmas em várias funções definidas posteriormente.

In [ ]:
```
from cryptography.hazmat.primitives.ciphers import (
    Cipher,algorithms,modes)
from cryptography.hazmat.primitives import padding
import os
import timeit
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np

sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]
results = {}
key = os.urandom(32) # key of 32 bytes (256 bits)
```

A função **encrypt(data,size,iv)** cria um objeto de cifra AES, no modo CBC (Cipher Block Chaining) com uma chave (key, variável global definida acima) e um vetor de inicialização (iv), para um ficheiro de tamanho size. Se os dados não forem múltiplos de 16 bytes, este adiciona preenchimento (padding PKCS7) para que fiquem com o tamanho correto. A função encripta os dados (data) e retorna o resultado.

A função **decrypt(ciphertext, size, iv)** cria um objeto de cifra AES, no mesmo modo, com a mesma chave e vetor de inicialização. Se os dados tiverem sido preenchidos anteriormente (padding), a função remove esse preenchimento após o processo. Retorna os dados originais desencriptados.

In [ ]:
```
def encrypt(data, size, iv):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    encryptor = cipher.encryptor()

    # bytes have to be a multiple of 16... if not add padding
    if size % 16 != 0:
```

```
        # Pad the data to the AES block size (128 bits)
        padder = padding.PKCS7(128).padder()
        padded_data = padder.update(data) + padder.finalize()
        ct = encryptor.update(padded_data) + (
            encryptor.finalize())
        return ct
    else:
        ct = encryptor.update(data) + encryptor.finalize()
        return ct


def decrypt(ciphertext, size, iv):
    # if not a size multiple of 16 it will have padding to be
    #removed
    if size % 16 != 0:
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
        decryptor = cipher.decryptor()
        padded_data = decryptor.update(ciphertext) + (
        decryptor.finalize())
        unpadder = padding.PKCS7(128).unpadder()
        data = unpadder.update(padded_data) + (
            unpadder.finalize())
        return data
    else:
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
        decryptor = cipher.decryptor()
        data = decryptor.update(ciphertext) + (
            decryptor.finalize())
        return data
```

Encriptação e Desencriptação com RSA

Para as funções de RSA são necessários alguns módulos da biblioteca cryptography, tal como um gerador de chave privada e chave pública, necessárias para a encriptação e desencriptação em RSA.

In [ ]:
```
from cryptography.hazmat.primitives.asymmetric import (
    rsa, padding)
from cryptography.hazmat.primitives import (
    serialization, hashes)

# Generate RSA Key Pair (2048-bit)
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)
public_key = private_key.public_key()
```

A função **encrypt_data(data)** usa uma chave pública (public_key) para cifrar os dados. Aplica OAEP (Optimal Asymmetric Encryption Padding) com SHA-256, que aumenta a segurança ao evitar ataques baseados em estrutura de

texto. Retorna os dados cifrados, que só podem ser decifrados com a chave privada (private_key) correspondente.

A função **decrypt_data(encrypted_data)** usa a chava privada (private_key) para decifrar os dados, aplicando o mesmo sistema OAEP + SHA-256 para garantir segurança. Retorna os dados originais, isto é, os dados anteriores à criptografia.

In [ ]:
```
# Function to encrypt data
def encrypt_data(data):
    return public_key.encrypt(
        data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

# Function to decrypt data
def decrypt_data(encrypted_data):
    return private_key.decrypt(
        encrypted_data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
```

Hashing com SHA-256

O processo hashing transforma o "input", file_data, de qualquer tamanho, num hash fixo de 256 bits, ou 64 caracteres hexadecimais. Para isto, é necessário importar a biblioteca hashlib. Também foi criada uma lista para armazenar os tamanhos que serão tratados com SHA-256 e um dicionário para armazenar o desvio padrão de cada tamanho.

In [ ]:
```
import hashlib
sizes = [8, 64, 512, 4096, 32768, 262144, 2097152]
std_dev = {}
#Gerar a hash SHA-256
def calculate_sha256_hash(file_data):
    return hashlib.sha256(file_data).hexdigest()
```

Cálculo de intervalos de confiança

Dá-nos uma estimativa de incerteza ao inferir sobre uma população com base numa amostra. Tem como nível de confiança 95%. com uma distribuição normal.

In [ ]:
```
# Calculate and return the confidence interval for the given data.
def get_confidence_interval(data, confidence=0.95):
    # Parameters: confidence (float):
    #The confidence level (default is 0.95).
    n = len(data)
    mean = np.mean(data)
    std_dev = np.std(data, ddof=1)
    se = std_dev / np.sqrt(n)

    # 95% confidence, using the normal distribution approximation:
    #the z-score is 1.96.
    z = 1.96
    margin_error = z * se
    return (mean - margin_error, mean + margin_error)
```

Medição do Tempo e processamento de ficheiros com AES

Para processar todos os ficheiros de um só tamanho (100), é usada a função **processAllFiles(size)**, que mede o tempo de encriptação e desencriptação de cada ficheiro, em microssegundos, que será impresso no terminal. Cada tempo será armazenado no respetivo array: se for o tempo de encriptação irá para o arrayEnc, e se for de desencriptação irá para o arrayDec. No final, armazenamos no dicionário results (variável global) esses arrays, com a chave "encryption_time" e "decryption_time" para facilitar a identificação. Também nos são dados os intervalos de confiança para cada tamanho de ficheiro.

In [ ]:
```
def processAllFiles(size):
    # Array to store encryption times for 100 files
    arrEnc = [0] * 100
    # Array to store decryption times for 100 files
    arrDec = [0] * 100
    for i in range(1, 101):
        file_path = os.path.join(
            "text_files", str(size), f"{size}_{i}.txt")
        with open(file_path, "rb") as file:
            plaintext = file.read()
        iv = os.urandom(16)  # IV has to be 16 bytes

        # Encrypt and time encryption
        encrypt_timer = timeit.Timer(
            lambda: encrypt(plaintext, size, iv))
        enc_time = (
            encrypt_timer.timeit(number=100) / 100) * 1000000
        arrEnc[i - 1] = enc_time
```

```
        ciphertext = encrypt(plaintext, size, iv)

        # Decrypt and time decryption
        decrypt_timer = timeit.Timer(
            lambda: decrypt(ciphertext, size, iv))
        dec_time = (
            decrypt_timer.timeit(number=100) / 100) * 1000000
        arrDec[i - 1] = dec_time

    confidenceEnc = get_confidence_interval(arrEnc)
    confidenceDec = get_confidence_interval(arrDec)
    print(f"{size} bytes:\tEncryption: (
          {confidenceEnc[0]:.2f},
          {confidenceEnc[1]:.2f})\tDecryption:
          ({confidenceDec[0]:.2f},
          {confidenceDec[1]:.2f})")
    # Store all times for the given size
    results[size] = {'encryption_time': arrEnc,
                     'decryption_time': arrDec}
```

A função **processUnique(file, size)** é utilizada para a medição de tempo de um só ficheiro, 100 vezes, para se observar as variações de tempo do mesmo a cada encriptação ou desencriptação. Armazenamos, assim, cada tempo de encriptação no arrayEnc e cada tempo de desencriptação no arrayDec, e no fim é chamada uma função **plot_results(arrayEnc, arrayDec, file)** que cria um gráfico de linhas com os resultados, exposto no capítulo 3, Análise de Resultados.

In [ ]:
```
def processUnique(file,size):
    file_path = os.path.join("text_files", str(size), file)
    # Check if the file exists
    if not os.path.isfile(file_path):
        print(f"Error: File {file_path} not found.")
        return
    # Read the file only once
    with open(file_path, "rb") as f:
        data = f.read()
    # Arrays to store encryption and decryption times
    arrayEnc = []
    arrayDec = []
    #print(f"{file:<8}:")
    for i in range(100):
        with open(file_path, "rb") as f:
            plaintext = f.read()
        iv = os.urandom(16)  # IV has to be 16 bytes
        # Encrypt and time encryption
        encrypt_timer = timeit.Timer(
            lambda: encrypt(plaintext, size, iv))
        enc_time = (
            encrypt_timer.timeit(number=1000) / 1000) * 1000000
        arrayEnc.append(enc_time)

        ciphertext = encrypt(plaintext, size, iv)
```

```
        # Decrypt and time decryption
        decrypt_timer = timeit.Timer(
            lambda: decrypt(ciphertext, size, iv))
        dec_time = (
            decrypt_timer.timeit(number=1000) / 1000) * 1000000
        arrayDec.append(dec_time)

    plot_results(arrayEnc, arrayDec, file)
```

Medição do Tempo e processamento de ficheiros com RSA

A função **process_all_files(size)**, usa o mesmo método que a função para AES, mas sem o vetor de inicialização. Também nos são fornecidos os intervalos de confiança para cada tamanho de ficheiro.

In [ ]:
```
def process_all_files(size):
    arrayEnc = [0]*100
    arrayDec = [0]*100
    results[size] = {'encryption_time': [], 'decryption_time': []}
    for i in range(1,101):
        file_path = os.path.join(
            "text_files", str(size), str(size)+"_"+str(i)+".txt")

        if os.path.isfile(file_path):
            with open(file_path, "rb") as f:
                data = f.read()

            # Measure encryption time
            enc_time = (timeit.timeit(
                lambda: encrypt_data(data),
                number=100) / 100)* (
                    1_000_000 )
            encrypted_data = encrypt_data(data)
            arrayEnc[i-1] = enc_time
            # Measure decryption time
            dec_time = (
                timeit.timeit(
                    lambda: decrypt_data(encrypted_data),
                    number=100)
                    / 100)* (1_000_000)
            arrayDec[i-1] = dec_time
            # Store results
            results[size]['encryption_time'].append(enc_time)
            results[size]['decryption_time'].append(dec_time)
            filename = str(size) + "_" + str(i) + ".txt"
            #print(f"{filename:<8} | {size:<12} | {enc_time:.9f}
            #{dec_time:.6f}")

    confidenceEnc = get_confidence_interval(arrayEnc)
    confidenceDec = get_confidence_interval(arrayDec)
```

```
    print(f"{size} bytes:\tEncryption: (
          {confidenceEnc[0]:.2f},
          {confidenceEnc[1]:.2f})\tDecryption: (
            {confidenceDec[0]:.2f},
            {confidenceDec[1]:.2f})")
```

A função **process_unique(file, size)** tem o mesmo efeito que a função processUnique para AES, imprimindo no terminal o tempo que o mesmo ficheiro demora a codificar e descodificar, para cada iteração. Após isso é chamada uma função **plot_results(arrayEnc, arrayDec, file)** que retorna um gráfico de linhas com os resultados, também acessível no Capítulo 3, Análise de Resultados.

In [ ]:
```
def process_unique(file,size):
    file_path = os.path.join("text_files", str(size), file)
    # Check if the file exists
    if not os.path.isfile(file_path):
        print(f"Error: File {file_path} not found.")
        return
    # Read the file only once
    with open(file_path, "rb") as f:
        data = f.read()
    # Arrays to store encryption and decryption times
    arrayEnc = []
    arrayDec = []
    #print(f"{file:<8}:")
    for i in range(100):  # Run 100 times for accuracy
        try:
            # Measure encryption time
            enc_time = (
                timeit.timeit(
                    lambda: encrypt_data(data),
                    number=100)
                    / 100)* 1_000_000
            encrypted_data = encrypt_data(data)
        except Exception as e:
            print(f"Error encrypting {file}: {e}")
            continue
        arrayEnc.append(enc_time)
        try:
            # Measure decryption time
            dec_time = (
                timeit.timeit(
                    lambda: decrypt_data(encrypted_data),
                    number=100)
                    / 100)* (1_000_000)
        except Exception as e:
            print(f"Error decrypting {file}: {e}")
            continue
        arrayDec.append(dec_time)
    # Print averaged results
        #print(f"Encryption Time: {arrayEnc[i]:.9f} s | Decryption
        #{arrayDec[i]:.6f} s")
    plot_results(arrayEnc, arrayDec, file)
```

Medição do Tempo e processamento de ficheiros com SHA

A função **process_files(base_dir, size)**, semelhante às funções processAllFiles (AES) e process_all_files (RSA), calcula e retorna o tempo de hashing de 100 ficheiros de cada tamanho, em microssegundos. É nos disponibilizado, também, o intervalo de confiança e o desvio padrão para os resultados obtidos.

In [ ]:
```
def process_files(base_dir, size):
    size_dir = os.path.join(base_dir, str(size))
    hash_times = []

    for i in range(1, 101):
        file_path = os.path.join(size_dir, f"{size}_{i}.txt")
        if os.path.exists(file_path):
            with open(file_path, "rb") as file:
                file_data = file.read()

                timer = timeit.Timer(
                    lambda: calculate_sha256_hash(file_data))
                hash_time = (
                    timer.timeit(number=100) / 100) * 1000000
                hash_times.append(hash_time)

    confidenceHash = get_confidence_interval(hash_times)
    print(f"{size} bytes:\tHashing:
          ({confidenceHash[0]:.2f}, {confidenceHash[1]:.2f})")
    # ddof=1 for sample std deviation
    std_hash = np.std(hash_times, ddof=1)
    std_dev[size] = std_hash
    # Gráfico de distribuição
    plt.figure(figsize=(10, 6))
    plt.text(0.75, 1.05, f"Std Dev: {std_hash:.2f}",
             fontsize=12, color='blue',
             verticalalignment='bottom',
             horizontalalignment='center',
             transform=plt.gca().transAxes)
    # Show ticks every 10 iterations
    tick_positions = range(0, 101, 10)
    # Set X-axis to display only 10th iterations
    plt.xticks(tick_positions)
    # Set the width of the bars
    bar_width = 1
    index = range(1, len(hash_times) + 1)  # Indices for X-axis
    # Plot the lines
    plt.plot([i - bar_width / 2 for i in index],
             hash_times, '-', linewidth=2,
             color='blue', label='Tempo de Hash')
    plt.ylabel('Tempo de hash (µs)')
    plt.xlabel('Iterações')
```

```
    plt.legend(loc='upper right')
    plt.title(f'Tempos de Hashing SHA ({size} bytes)')
    #plt.savefig(f'{file_name}_performance.png', dpi=120)
    plt.show()
    plt.close()

    return sum(hash_times)/len(hash_times) if hash_times else None
```

A função **process_unique_file(file_name, size)**, da mesma forma que as funções para AES e RSA, calcula e retorna o tempo de hashing de um só ficheiro iterado 100 vezes, em microssegundos. Para além disso, retorna um gráfico de linhas associado aos resultados obtidos, com o desvio padrão dos resultados.

In [ ]:
```
def process_unique_file(file_name, size):
    base_dir = find_correct_path()
    file_path = os.path.join(
        base_dir, str(size), file_name)

    if not os.path.exists(file_path):
        print(f"Arquivo {file_path} não encontrado!")
        return

    with open(file_path, "rb") as f:
        data = f.read()

    hash_times = []
    for _ in range(100):
        timer = timeit.Timer(
            lambda: calculate_sha256_hash(data))
        hash_time = (
            timer.timeit(number=100) / 100) * 1000000
        hash_times.append(hash_time)

    # Gráfico de distribuição
    plt.figure(figsize=(10, 6))
    # ddof=1 for sample std deviation
    std_hash = np.std(hash_times, ddof=1)
    # Gráfico de distribuição
    plt.figure(figsize=(10, 6))
    plt.text(0.75, 1.05, f"Std Dev: {std_hash:.2f}",
             fontsize=12, color='blue',
             verticalalignment='bottom',
             horizontalalignment='center',
             transform=plt.gca().transAxes)
    # Show ticks every 10 iterations
    tick_positions = range(0, 101, 10)
    # Set X-axis to display only 10th iterations
    plt.xticks(tick_positions)
    # Set the width of the bars
    bar_width = 1
    # Indices for X-axis
    index = range(1, len(hash_times) + 1)
    # Plot the lines
```

```
    plt.plot([i - bar_width / 2 for i in index],
             hash_times, '-', linewidth=2,
             color='blue', label='Tempo de Hash')
    plt.ylabel('Tempo de hash (µs)')
    plt.xlabel('Iterações')
    plt.legend(loc='upper right')
    plt.title(f'Distribuição de tempos - {file_name}')
    #plt.savefig(f'{file_name}_performance.png', dpi=120)
    plt.show()
    plt.close()
```

Funções de projeção dos resultados em gráficos (AES, RSA, SHA)

As funções são semelhantes, logo serão generalizadas para os três modos.

A função **plot_results(arrayEnc, arrayDec, file)** retorna um gráfico de linhas com os tempos de encriptação e desencriptação respetivos a cada iteração de um só ficheiro. O tempo de encriptação está a azul, e o tempo de desencriptação a vermelho. O tempo é medido em microssegundos. Também é possível ter acesso ao desvio padrão (std dev) dos resultados obtidos. A função apresentada é para o modo AES, sendo igual nos restantes modos.

In [ ]:
```
def plot_results(arrayEnc, arrayDec, file):
    # Create a figure
    plt.figure(figsize=(12, 8))

    # Set the width
    bar_width = 1
     # Indices for X-axis
    index = range(1, len(arrayEnc) + 1)

    # Plot the lines
    plt.plot([i - bar_width / 2 for i in index],
             arrayEnc, '-',
             linewidth=2, color='blue',
             label='Encryption Time')
    plt.plot([i - bar_width / 2 for i in index],
             arrayDec, '-',
             linewidth=2, color='red',
             label='Decryption Time')

    # Compute standard deviation for encryption times
    # ddof=1 for sample std deviation
    std_enc = np.std(arrayEnc, ddof=1)
    std_dec = np.std(arrayDec, ddof=1)

    # Adding labels and title
    plt.xlabel('Iteration')
    plt.ylabel('Time (microseconds)')
    plt.title(f"AES Encryption and Decryption Times for
```

```
            {file}")

    # Set X-axis labels to the index of iterations
     # Show ticks every 10 iterations
    tick_positions = range(0, 101, 10)
    # Set X-axis to display only 10th iterations
    plt.xticks(tick_positions)
    plt.legend(loc='upper right')

    # Display both standard deviations
    #below the title and above the bars
    plt.text(0.25, 1.05, f"Std Dev (Encryption):
             {std_enc:.2f}",
             fontsize=12, color='blue',
             verticalalignment='bottom',
             horizontalalignment='center',
             transform=plt.gca().transAxes)

    plt.text(0.75, 1.05, f"Std Dev (Decryption):
             {std_dec:.2f}",
             fontsize=12, color='red',
             verticalalignment='bottom',
             horizontalalignment='center',
             transform=plt.gca().transAxes)
    # Display the plot
    plt.show()
```

A função **plot_graph(results)**, especificada para o modo AES, é utilizada para retornar o gráfico de linhas para um tamanho específico, com as variações dos 100 ficheiros correspondentes. É possível, também, observar o desvio padrão dos resultados.

In [ ]:
```
def plot_graph(results):
    # Loop através dos resultados para cada pasta/tamanho
    for folder_size, times in results.items():
        # Criar um novo gráfico para cada tamanho de pasta
        # Definir o tamanho da figura
        plt.figure(figsize=(12, 8))

        bar_width = 1
        # Indices for X-axis
        index = range(1, len(times['encryption_time']) + 1)
        # Plotando o tempo de encriptação
        plt.plot([i - bar_width / 2 for i in index],
                 times['encryption_time'], '-',
                 linewidth=2, color='blue',
                 label='Encryption Time')

        # Plotando o tempo de decriptação
        plt.plot([i - bar_width / 2 for i in index],
                 times['decryption_time'], '-',
                 linewidth=2, color='red',
                 label='Decryption Time')

        # Calculate the standard deviation
```

```
        #for encryption and decryption times
        std_enc = np.std(times['encryption_time'], ddof=1)
        std_dec = np.std(times['decryption_time'], ddof=1)

        # Ajustar os rótulos e título para o gráfico
        plt.xlabel('File Index')
        plt.ylabel('Time (Microseconds)')
        plt.title(f"Encryption and Decryption Times for AES (
                  {folder_size} bytes)")

        # Definir os ticks no eixo X para mostrar a cada 10 arquivo
        # Mostrar ticks a cada 10 iterações
        tick_positions = range(0, 101, 10)
        plt.xticks(tick_positions)
        plt.legend()
        # Display both standard deviations
        #below the title and above the bars
        plt.text(0.25, 1.05, f"Std Dev (Encryption):
                 {std_enc:.2f}",
                 fontsize=12, color='blue',
                verticalalignment='bottom',
                horizontalalignment='center',
                transform=plt.gca().transAxes)

        plt.text(0.75, 1.05, f"Std Dev (Decryption):
                 {std_dec:.2f}",
                 fontsize=12, color='red',
                verticalalignment='bottom',
                horizontalalignment='center',
                transform=plt.gca().transAxes)
        plt.show()
```

A função **plot_graph_avg(results)**, utilizada para o ficheiro de RSA, tem como retorno o gráfico de linhas para a média de cada tamanho utilizado pelo modo. No caso do RSA, é nos dada a média de tempo de encriptação e decriptação para os tamanhos 2, 4, 8, 16, 32, 64 e 128 bytes. Uma função com a mesma estrutura foi utilizada para retornar um gráfico de linhas para a média de cada tamanho utilizado pelo modo AES. No caso do AES os tamanhos são de 8, 64, 512, 4096, 32768, 262144 e 2097152 bytes.

In [ ]:
```
def plot_graph_avg(results): # works well, corrigir avg dec
    # For each size, compute the average encryption and decryption
    sizes = sorted(results.keys())
    avg_enc_times = []
    avg_dec_times = []

    for s in sizes:
        avg_enc = sum(results[s]['encryption_time']) / (
            len(results[s]['encryption_time']))
        avg_dec = sum(results[s]['decryption_time']) / (
            len(results[s]['decryption_time']))
        # Calculate the standard deviation
        #for encryption and decryption times
        avg_enc_times.append(avg_enc)
```

```
        avg_dec_times.append(avg_dec)


    plt.figure(figsize=(12, 8))

    # Line plot for encryption and decryption times
    plt.plot(sizes, avg_enc_times, marker='o',
             color='blue', label='Encryption Time')
    plt.plot(sizes, avg_dec_times, marker='x',
             color='red', label='Decryption Time')

    # Adding labels and title
    plt.xscale('log')  # 'log' for better visualization
    plt.xlabel('Size (Bytes)')
    plt.ylabel('Time (Microseconds)')
    plt.title("Average Encryption and Decryption Times for RSA")
    plt.xticks(sizes, [str(x) for x in sizes])
    # Display the bars and the legend
    plt.legend()
    plt.show()
```

A função **plot_sha256_performance(results)** funciona como a plot_graph_avg(results), mas para hashing, retornando um gráfico de linhas.

In [ ]:
```
def plot_sha256_performance(results, std_devs):
    x_values = list(results.keys())
    y_values = list(results.values())

    plt.figure(figsize=(12, 8))
    plt.plot(x_values, y_values, '-',
             linewidth=2,color='blue', label='Tempo de Hash')
    plt.xscale('log')
    plt.xlabel('Tamanho do arquivo (bytes)', fontsize=12)
    plt.ylabel('Tempo médio de hash (µs)', fontsize=12)
    plt.title('Desempenho do SHA-256 por Tamanho de Arquivo',
              fontsize=14)
    plt.grid(True, alpha=0.3)
    plt.xticks(x_values, [str(x) for x in x_values],
               rotation=45)
    plt.legend(loc='upper right')
    #plt.savefig('sha256_performance.png', dpi=300)
    plt.show()
    plt.close()
```

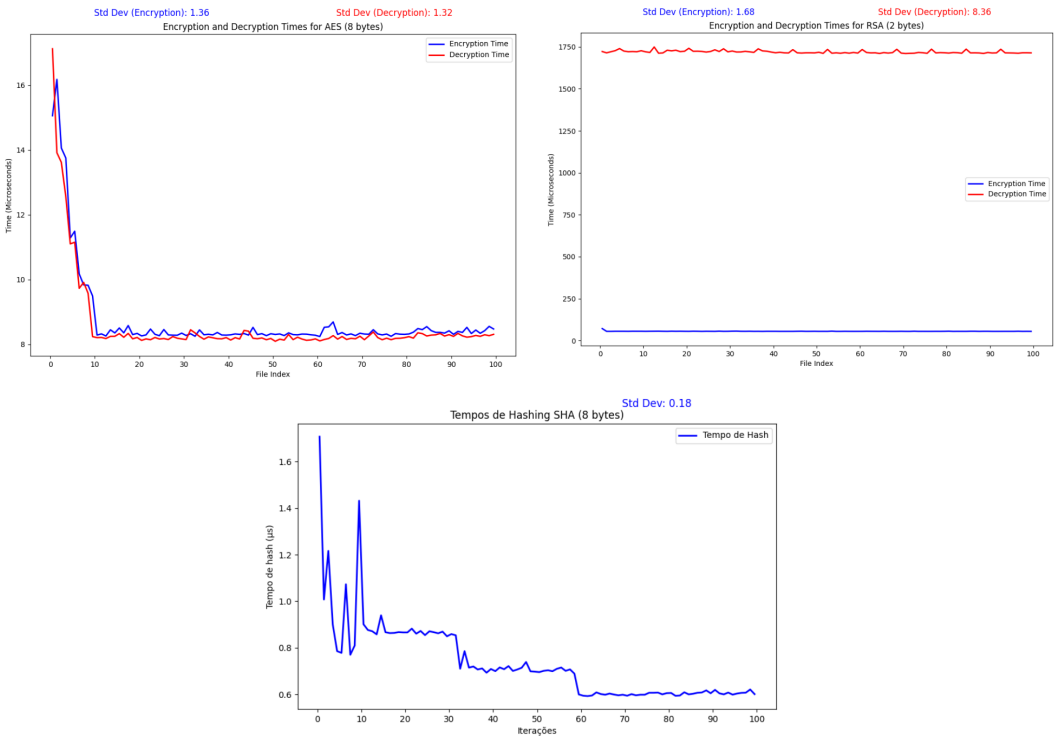
---

3. Análise de Resultados

> O computador usado para a obtenção dos resultadoa abaixo é o MacBook Air 2022, de chip M2. Possui o OS Sequoia 15.3.1, memória de 16GB e tem como disco de arranque o Macintosh HD.

Na seguinte análise dos resultados obtidos serão comparados os tempos de cada modo em relação ao tamanho do ficheiro, isto é, separando os tamanhos trabalhados entre pequenos, médios e grandes. Assim, será feita uma comparação direta entre modos criptográficos, tal como análises individuais. As análises estarão focadas nas últimas iterações do processo, pois as primeiras não são relevantes e pouco credíveis.

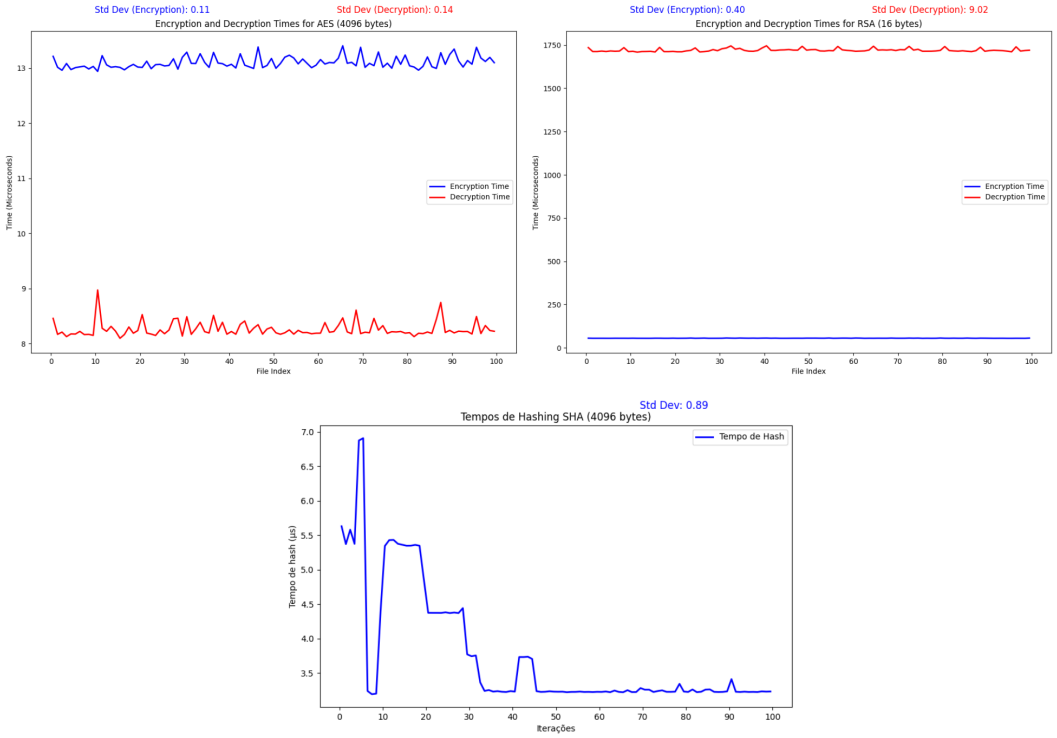
Ficheiros pequenos: comparação entre os resultados dos 3 modos



Em relação aos ficheiros pequenos, foram selecionados os menores tamanhos a serem trabalhados: 2 bytes para o modo RSA, e 8 bytes para os restantes modos. Com 8 bytes, o modo AES cifra e decifra com idêntica rapidez, rondando os 10 µs. Os respetivos desvios padrão são assim semelhantes (1,36 µs e 1,32 µs, respetivamente). Já o modo RSA cifrou os ficheiros de 2 bytes em 100 µs mas decrifrou-os em 1750 µs, o que demonstra que é mais rápido encriptar um ficheiro do que desencriptá-lo. Os desvios padrão coincidem com isso, sendo de 1.68 µs e 8.36 µs, respetivamente. Por fim, no processo de Hashing (SHA-256), é possível ver uma ligeira descida de tempo, estagnando aproximadamente nos 0.6 µs. O seu desvio padrão é igualmente pequeno, com 0.18 µs. Permite-nos assim concluir que, para ficheiros pequenos, o modo SHA é bastante rápido. Numa análise individual, a codificação e descodificação de vários ficheiros com um único tamanho pode ter as suas variações. No modo AES, não possui quase variações de tempo, excluindo os primeiros dez ficheiros, tal como no modo RSA. Já o modo SHA

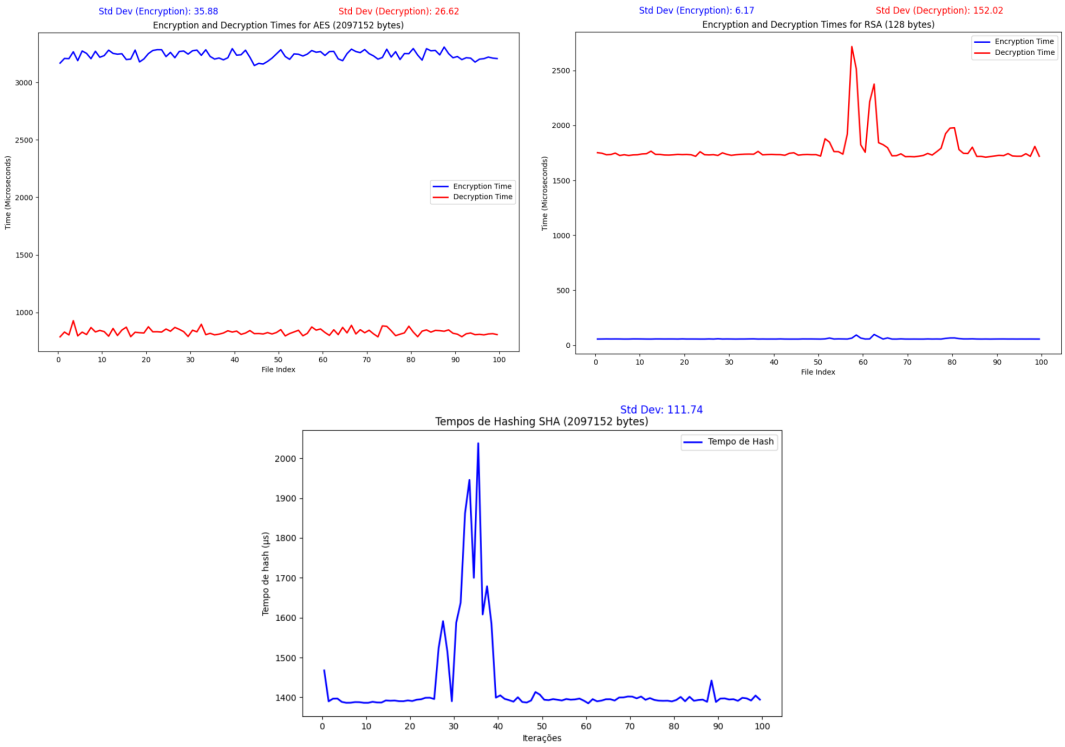
desce ao longo das iterações, tendo um início irrelevante acima de 1.6 µs, e desce até aos 0.6 µs. Conclui-se assim que, para ficheiros pequenos, a variação é quase nula.

Ficheiros médios: comparação entre os resultados dos 3 modos



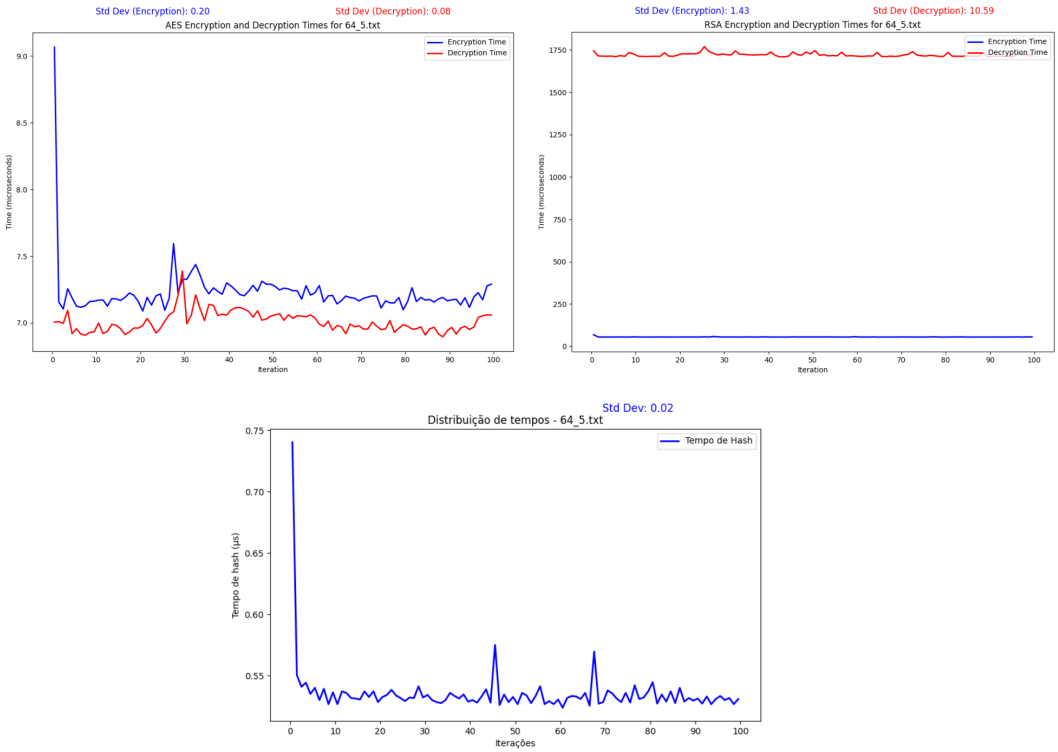
No que diz respeito a ficheiros médios, foram escolhidos para análise: 16 bytes para o RSA e 4096 bytes para os restantes modos. Com AES, o tempo de codificação, que ronda os 13.1 µs, é maior que o tempo de descodificação, aproximadamente 8.5 µs, sendo ambos estáveis e com um desvio padrão baixo, de 0.11 e 0.14, respetivamente. No modo RSA, o tempo mantém-se igual ao de um ficheiro pequeno, isto é, cifra em 100 µs e decifra em 1750 µs. Assim, os desvios padrão respetivos coincidem com as conclusões anteriores, com 0.40 µs de encriptação e 9.02 de desencriptação. Por último, no modo SHA, tal como num ficheiro pequeno, ocorrem variações e descidas de tempo. Descartando as variações iniciais, que chegam quase a 7 µs, a partir do 50º ficheiro o tempo de hashing é de aproximadamente 2 µs. O desvio padrão dos resultados obtidos é baixo, com um valor de 0.89 µs. Conclui-se que no geral, e excluindo as variações iniciais, o tempo é estável. Pode-se concluir que a codificação e descodificação de vários ficheiros com um único tamanho continua a não ter muitas variações.

Ficheiros grandes: comparação entre os resultados dos 3 modos



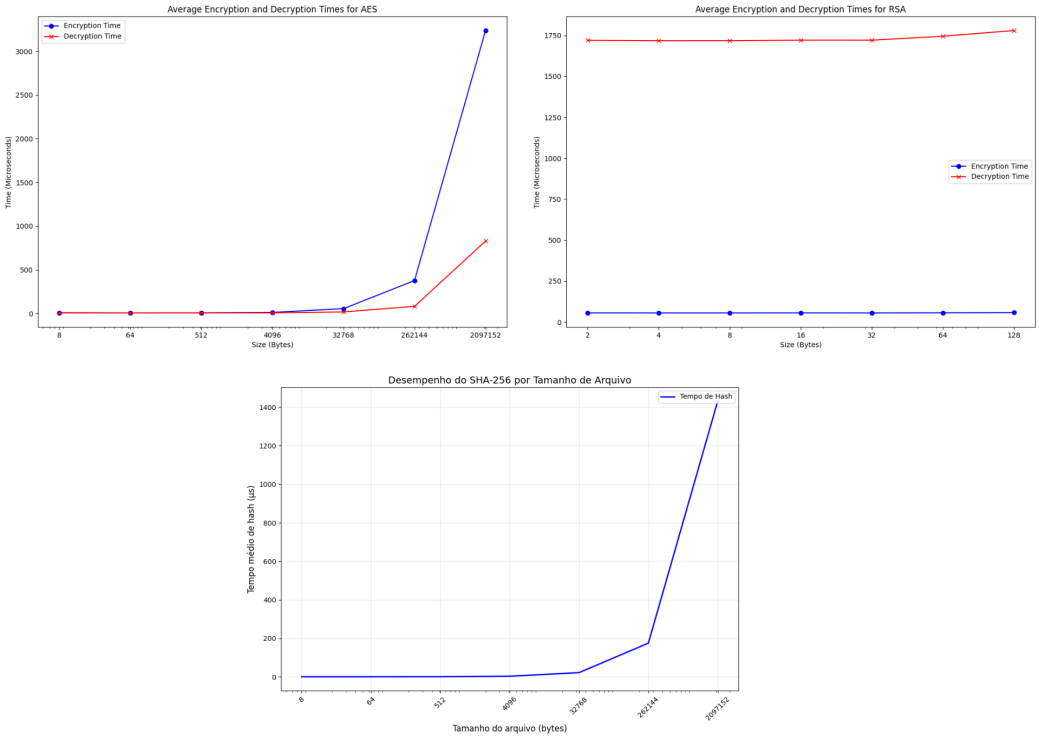
Relativamente a ficheiros grandes, serão analisados os tamanhos: 128 bytes para o RSA e 2097152 bytes para os restantes modos. Usando o AES, ambos os tempos, comparativamente aos tamanhos anteriores, aumentaram significativamente. O tempo de encriptação aproxima-se dos 4000 µs, e o tempo de desencriptação é de mais de 500 µs, havendo assim uma grande diferença entre tempos, o que indica que os processos envolvidos na encriptação podem ser mais exigentes que na desencriptação. Com o modo RSA, os tempos continuam semelhantes aos analisados anteriormente. Neste caso, ocorreram variações no 60º ficheiro, com um pico máximo de descodificação de aproximadamente 2700 µs, e por isso o seu desvio padrão é alto, com 152.02 µs. O desvio padrão de codificação continua baixo, com 6.17 µs. Por último, o SHA possui uma variação grande no 35º ficheiro, por razões comuns relacionadas possivelmente ao hardware, atingindo uma Alor de mais de 2000 µs. O desvio padrão reflete nessa variação, com o valor de 111.74 µs. De resto, os tempos são baixos e estáveis, de aproximadamente 1400 µs. É possível concluir que, tal como nos tamanhos inferiores, em tamanhos grandes a iteração entre ficheiros do mesmo tamanho não possui grandes alterações, a não ser que, por alguma razão, a performance do CPU se altere, por exemplo.

Ficheiro único (64_5.txt): comparação entre os resultados dos 3 modos



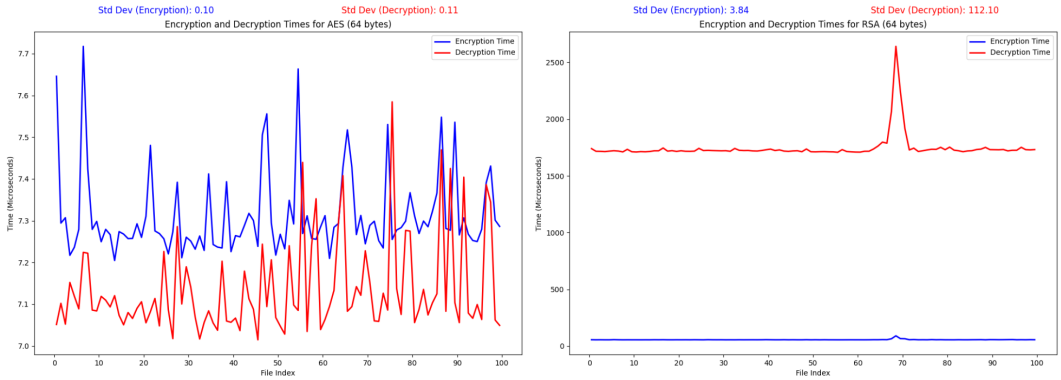
Para a análise das variações de tempo num ficheiro único, iterado 100 vezes, foi escolhido o ficheiro 64_5.txt, de tamanho 64 bytes. Com o modo AES, a variação é quase nula, com tempos de codificação e descodificação que se encontram entre 6.8 µs e 7.6 µs. Isso reflete-se no desvio padrão, de valores 0.20 e 0.08 respetivamente. Usando o RSA, as variações são pequenas, situando-se principalmente na desencriptação. O tempo de encriptação é de 100 µs, como nos restantes tamanhos, e o tempo de desencriptação é de 1750 µs. Os desvios padrão são 1.43 µs e 10.59 µs, respetivamente. Como nos modos anteriores, o modo SHA não possui grandes variações, com tempos situados entre 0.53 µs e 0.58 µs, com um desvio padrão de 0.02 µs. É, então, o modo com menos variações.

Média de tempos: comparação entre os resultados dos 3 modos



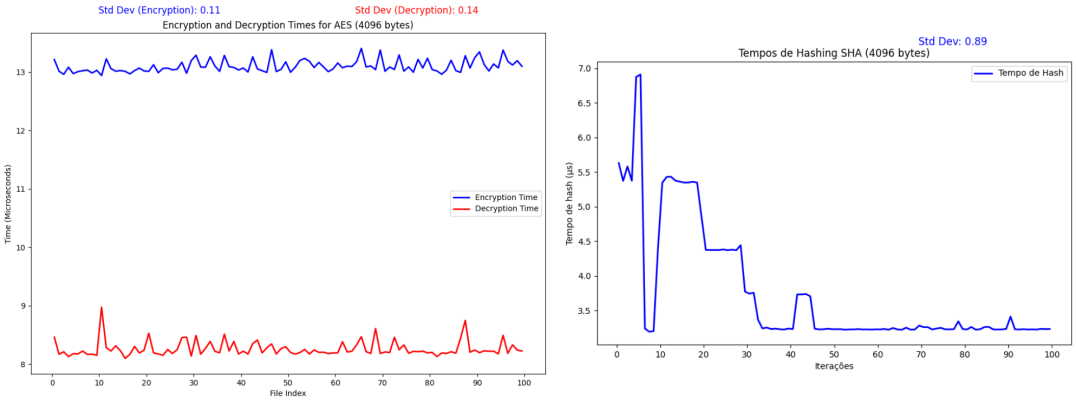
Numa análise geral à média dos resultados para cada modo, podem-se retirar algumas conclusões. No gráfico de AES, observa-se um aumento quase exponencial no tempo de encriptação, e um aumento linear no tempo de desencriptação. Pode-se concluir que o tamanho dos arquivos influencia, de certa forma, os tempos de encriptação e desencriptação. O valor mais baixo é muito próximo de 0 µs em ambos os processos. O valor mais alto é de mais de 3000 µs para a encriptação, e de quase 1000 µs para a desencriptação. No gráfico de RSA, é possível observar um crescimento constante em ambos os processos. Na codificação, o tempo mantém-se, para cada tamanho, nos 100 µs. Na descodificação ronda os 1750 µs. É possível concluir que, para os tamanhos estudados, estes não influenciam o tempo de encriptação e decriptação, visto que numa chave de 2048 bits, ou 256 bytes, todos os tamanhos são menores que a mesma. Em tamanhos muito maiores, os tempos iriam aumentar significativamente. Por último, no gráfico de SHA, é visível um crescimento potencialmente exponencial, que atinge os 1400 µs com o tamanho máximo de 2097152 bytes. O tempo mais baixo obtido foi de quase 0 µs. Conclui-se assim que o tamanho de um ficheiro influencia o tempo de hashing.

Comparação entre encriptação AES e encriptação RSA



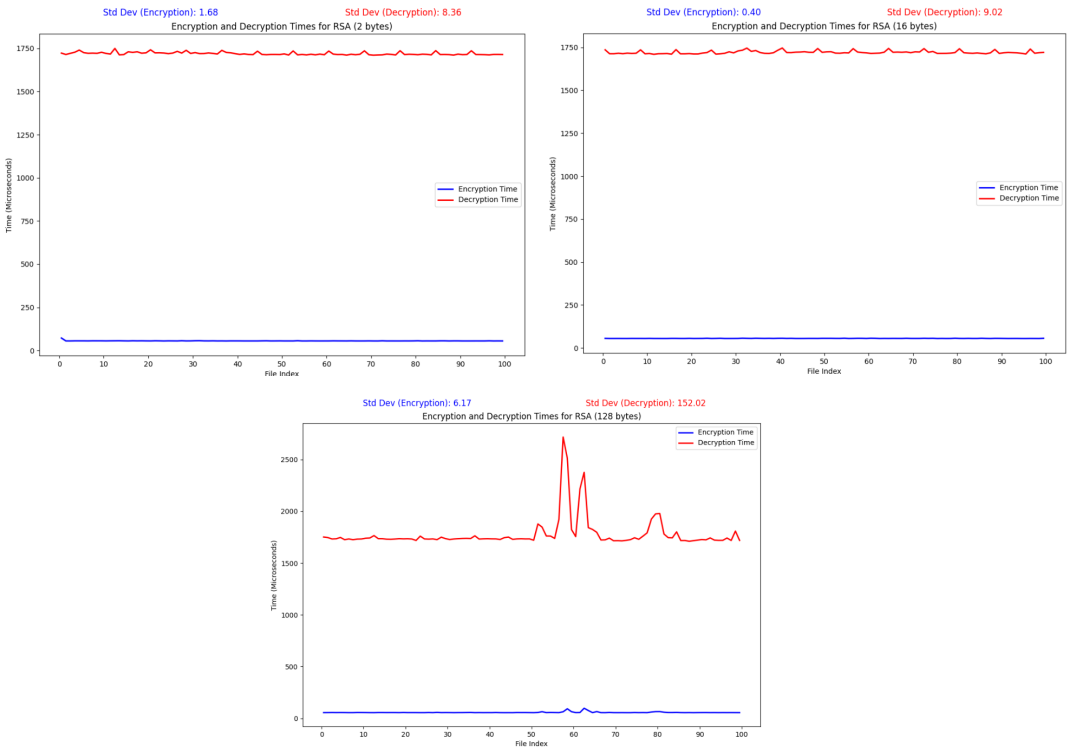
A encriptação é um aspeto fundamental da segurança digital, protegendo informação de acessos não autorizados. Entre vários modelos de encriptação, o AES e o RSA são dois métodos proeminentes que representam dois tipos fundamentais de algoritmos criptográficos. Têm objetivos distintos, cada um com as suas vantagens e desvantagens. A escolha entre estes dois métodos depende da necessidade do utilizador. Para uma encriptação segura, eficiente e linear de grandes quantidades de dados, o AES é preferível. Por outro lado, em situações que requerem comunicações seguras entre canais com pouca segurança, como a Internet, o RSA providencia um método seguro para a troca de chaves, que podem ser posteriormente usadas com o AES. Em relação à velocidade, o AES é muito mais rápido que o RSA, e é mais adequado aquando de criptografar grandes quantidades de dados. No exemplo dado, num ficheiro de 64 bytes, a encriptação é bastante rápida no AES, estando entre os 7.2 µs e os 7.7 µs, com um desvio padrão de 0.10 µs. Já no RSA, ronda os 100 µs, com um desvio padrão de 3.84 µs. A diferença é grande, o que comprova que a encriptação de AES é mais eficiente que a encriptação de RSA. O modo RSA é tipicamente usado para a troca segura de chaves e assinaturas digitais, pois lida com tamanhos de dados bastante pequenos. O modo AES é usado para a encriptação de dados com tamanhos variados, principalmente em grande escala.

Comparação entre encriptação AES e digest generation SHA-256



Os algoritmos criptográficos AES e SHA-256 servem diferentes propósitos e têm características distintas. A grande diferença entre estes algoritmos é que o SHA-256 não possui um processo de encriptação, mas sim um processo único, com apenas um sentido. É usado em ficheiros grandes para computar o seu "hash" ou "digest", e com esse resultado não é possível recuperar o ficheiro original. Já o AES é um algoritmo de criptografia simétrica, que usa blocos de 16 bytes e os cifra, com uma chave única que permite a codificação e a descodificação. Comparando a encriptação AES com o tempo de hashing do SHA-256, é possível ver pequenas diferenças, e concluir que o processo de hashing é ligeiramente mais rápido que a encriptação AES nos resultados obtidos. No exemplo dado, num ficheiro de 4096 bytes, a encriptação AES tem um tempo de aproximadamente 13.3 µs, com um desvio padrão de 0.11 µs. Já no modo SHA-256, ignorando as variações iniciais das primeiras 30 iterações, tem um tempo de 3 µs, e um desvio padrão de 0.89 µs. Mesmo que o desvio padrão seja maior no SHA-256 do que no AES, isso só ocorre devido às variações iniciais do SHA-256, que chegam aos 6.9 µs. Mesmo assim, o algoritmo AES é melhor que o SHA-256, pois este reversível com a chave correta.

Comparação entre o tempo de encriptação e decriptação de RSA



Para o RSA, especificamente, a desencriptação é mais lenta que a encriptação. Isto acontece porque tanto a encriptação como a desencriptação usam exponenciais modulares, mas, enquanto que o expoente público de encriptação é tipicamente pequeno e fixo (está normalmente entre 3 e

65537), o expoente privado de desencriptação é quase tão grande como o próprio módulo. Por isso, duplicar o tamanho do módulo irá duplicar o tempo de encriptação e irá quadriplicar o tempo de desencriptação. Uma solução em alguns casos é recorrer ao uso do Teorema Chinês do Resto (Chinese Remainder Theorem), que acelera a desencriptação do RSA. Estudando os resultados obtidos, nos ficheiros de 2, 16 e 128 bytes, a diferença é mínima. Isto deve-se ao tamanho da chave pública, de 2048 bits. Os tamanhos trabalhados são pequenos o suficiente para se encaixar num único bloco (256 bytes). Mas comparando o tempo de codificação com o de descodificação, a diferença é grande. Nesses ficheiros, o tempo de codificação é de 100 µs, com um desvio padrão de 1.68 µs, e o tempo de descodificação é de 1750 µs, com um desvio padrão de 8.36 µs, o que comprova a informação acima.

Intervalos de Confiança

```
Confidence Intervals (microseconds) for AES:
------------------------------------------------------------------------
8 bytes:        Encryption: (8.47, 9.00)        Decryption: (8.31, 8.83)
64 bytes:       Encryption: (7.29, 7.33)        Decryption: (7.11, 7.16)
512 bytes:      Encryption: (8.10, 8.14)        Decryption: (7.37, 7.40)
4096 bytes:     Encryption: (13.08, 13.12)      Decryption: (8.24, 8.29)
32768 bytes:    Encryption: (55.54, 56.11)      Decryption: (18.17, 18.77)
262144 bytes:   Encryption: (376.55, 378.25)    Decryption: (81.10, 82.47)
2097152 bytes:  Encryption: (3229.70, 3243.77)  Decryption: (823.74, 834.18)
```

```
Confidence Intervals (microseconds) for RSA:
------------------------------------------------------------------------
2 bytes:        Encryption: (55.38, 56.04)      Decryption: (1718.17, 1721.45)
4 bytes:        Encryption: (55.25, 55.75)      Decryption: (1715.43, 1719.50)
8 bytes:        Encryption: (55.34, 55.66)      Decryption: (1715.40, 1719.65)
16 bytes:       Encryption: (55.56, 55.72)      Decryption: (1718.99, 1722.52)
32 bytes:       Encryption: (55.46, 55.60)      Decryption: (1718.95, 1722.38)
64 bytes:       Encryption: (55.49, 57.00)      Decryption: (1722.97, 1766.92)
128 bytes:      Encryption: (56.39, 58.81)      Decryption: (1749.94, 1809.53)
```

```
Confidence Intervals (microseconds) for SHA:
------------------------------------------------
8 bytes:        Hashing: (0.70, 0.77)
64 bytes:       Hashing: (0.76, 0.81)
512 bytes:      Hashing: (1.08, 1.16)
4096 bytes:     Hashing: (3.58, 3.93)
32768 bytes:    Hashing: (22.09, 22.31)
262144 bytes:   Hashing: (174.25, 176.52)
2097152 bytes:  Hashing: (1408.90, 1452.70)
```

O intervalo de confiança (IC) é uma faixa de valores que provavelmente contém o valor real de um parâmetro populacional, como a média. Foi usado um IC de 95%, o que significa que, se repetirmos a experiência várias vezes, 95% dos intervalos irão conter o valor real. A baixo estão os intervalos de confiança dos 3 algoritmos criptográficos. Um exemplo de um intervalo de confiança para encriptação, para o modo AES, com ficheiros de tamanho 512 bytes, é (8.10, 8.14). Isso significa que há 95% de certeza de que a média populacional está dentro desse intervalo.

---

4. Conclusão

Através este trabalho, é possível concluir que o tamanho dos arquivos influencia diretamente o tempo de processamento dos algoritmos criptográficos. Enquanto que o RSA mantém tempos relativamente constantes em ficheiros pequenos, apresenta um crescimento significativo em relação ao tempo de processamento de ficheiros maiores, tornando-se ineficiente. Por outro lado, o AES demonstra um desempenho altamente linear, mantendo tempos de encriptação e desencriptação significativamente pequenos, mesmo em tamanhos grandes. Os testes realizados com o algoritmo SHA-256 demonstram que o tempo de processamento do mesmo depende diretamente do tamanho da entrada, uma vez que o algoritmo processa os dados em blocos de 512 bits. Embora seja altamente seguro e eficiente para garantir a integridade e autenticidade dos dados, o seu desempenho pode ser impactado em cenários onde grandes quantidades de informação precisam ser processadas rapidamente. Os resultados obtidos confirmam, então, que os algoritmos simétricos, como o AES, são mais eficientes para encriptação de grandes quantidades de dados, enquanto que algoritmos assimétricos, como o RSA, são mais adequados para encriptar pequenas quantidades de dados, como chaves. Esta diferença de desempenho reforça o método de combinar algoritmos. Assim, o estudo reforça a importância da escolha adequada do algoritmo de criptografia dependendo do contexto e da necessidade, equilibrando segurança e eficiência para otimizar o desempenho dos sistemas criptográficos.

---

5. Webgrafia

- OpenAI (2025). ChatGPT. [online] chatgpt.com. Available at: https://chatgpt.com;
- Wikipedia Contributors (2024). RSA (cryptosystem). [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/RSA_cryptosystem;
- Das, D. (2023). Difference Between AES and SHA256. [online] Medium. Available at: https://diptendud.medium.com/difference-between-aes-and-sha256-706d6b2eb2ef;
- Why (2013). Why is RSA encryption significantly faster than decryption? [online] Cryptography Stack Exchange. Available at: https://crypto.stackexchange.com/questions/6378/why-is-rsa-encryption-significantly-faster-than-decryption [Accessed 2 Apr. 2025];
- key, R. (2013). PGP RSA key size - encryption/decryption time. [online]

Information Security Stack Exchange. Available at: https://security.stackexchange.com/questions/41937/pgp-rsa-key-size-encryption-decryption-time [Accessed 2 Apr. 2025];

- Team, T.P. (2024). AES vs RSA Encryption -. [online] Phalanx.io. Available at: https://phalanx.io/aes-vs-rsa-encryption/.