

# 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).

## 1. Introdução

O objetivo do seguinte projeto é medir o tempo que AES, RSA e SHA levam para processar arquivos de diferentes tamanhos, usando uma implementação em Python para os mecanismos de encriptação/descriptação e hash.

### AES (Advanced Encryption Standard)

O AES é um algoritmo de criptografia simétrica, ou seja, usa a mesma chave para encriptação e descriptação. 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 descriptação. 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 transmissão 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.

O trabalho especifica o uso do SHA-256, que gera um hash de 256 bits. É amplamente utilizado em assinaturas digitais, blockchain e verificação de integridade de arquivos.

---

## 2. Implementação

### Geração de Arquivos

Foram gerados, para este projeto, 100 ficheiros para cada tamanho.

```
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) # Ensure folder exists

    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 Descrição com AES

Bibliotecas e módulos necessários para a implementação das funções de AES, e variáveis globais utilizadas nas mesmas:

```
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) 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 descriptografados.

```
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 Descrição com RSA

Para as funções de RSA são necessários alguns módulos, tal como um gerador de chave privada e chave pública, necessárias para a encriptação e decriptação 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 criptografar 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 criptografados, que só podem ser descriptografados com a chave privada (private\_key) correspondente.

A função **decrypt\_data(encrypted\_data)** usa a chave privada (private\_key) para descriptografar 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 foram 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**, que nos dá uma estimativa de incerteza ao inferir sobre uma população com base numa amostra. Têm como nível de confiança 95%. numa 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
            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:
            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 decriptação de cada ficheiro, em microsegundos, 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 decriptação 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.

```
In [ ]: def processAllFiles(size):
    arrEnc = [0] * 100 # Array to store encryption times for 100 files
    arrDec = [0] * 100 # Array to store decryption times for 100 files
    for i in range(1, 101):
        file_path = os.path.join("text_files", str(size), f"{size}_file_{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}), {confidenceDec[0]:.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 um só ficheiro, 100 vezes, para se observar as variações de tempo de um só ficheiro a cada encriptação ou decriptação. Armazenamos assim cada tempo de encriptação no arrayEnc e cada tempo de decriptação 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)
enc_time = (encrypt_timer.timeit(number=1000) / 1000) * 1000
arrayEnc.append(enc_time)

ciphertext = encrypt(plaintext, size, iv)

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

```

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))

        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=1000))
                encrypted_data = encrypt_data(data)
                arrayEnc[i-1] = enc_time
                # Measure decryption time
                dec_time = (timeit.timeit(lambda: decrypt_data(encrypted_data), number=1000))
                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:.9f}")

    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 criptografar e descriptografar, 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))
            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))
        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 Time: {arrayDec[i]:.9f} 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(
            hash_time = (timer.timeit(number=100) / 100) * 1000
            hash_times.append(hash_time)

confidenceHash = get_confidence_interval(hash_times)
print(f"{size} bytes:\tHashing: ({confidenceHash[0]:.2f}, {conf

std_hash = np.std(hash_times, ddof=1) # ddof=1 for sample std
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, c
        verticalalignment='bottom', horizontalalignment='cente
tick_positions = range(0, 101, 10) # Show ticks every 10 itera
plt.xticks(tick_positions) # Set X-axis to display only 10th i
# 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, '-', l
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))
std_hash = np.std(hash_times, ddof=1) # ddof=1 for sample std
# Gráfico de distribuição
plt.figure(figsize=(10, 6))
plt.text(0.75, 1.05, f"Std Dev: {std_hash:.2f}", fontsize=12, color='red',
         verticalalignment='bottom', horizontalalignment='center')
tick_positions = range(0, 101, 10) # Show ticks every 10 iterations
plt.xticks(tick_positions) # Set X-axis to display only 10th iterations
# 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=1, color='blue')
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 deciptação respectivos a cada iteração de um só ficheiro. O tempo de encriptação está a azul, e o tempo de deciptação a vermelho. O tempo é medido em microsegundos. Também é possível ter acesso ao desvio padrão (std) 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
        index = range(1, len(arrayEnc) + 1) # Indices for X-axis

        # Plot the lines
        plt.plot([i - bar_width / 2 for i in index], arrayEnc, '-', linewidth=1, color='blue')
        plt.plot([i - bar_width / 2 for i in index], arrayDec, '-', linewidth=1, color='red')

        # Compute standard deviation for encryption times

```

```

std_enc = np.std(arrayEnc, ddof=1) # ddof=1 for sample std dev.
std_dec = np.std(arrayDec, ddof=1) # ddof=1 for sample std dev.

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

# Display both standard deviations below the title and above the plot
plt.text(0.25, 1.05, f"Std Dev (Encryption): {std_enc:.2f}", fontweight='bold',
        verticalalignment='bottom', horizontalalignment='center')

plt.text(0.75, 1.05, f"Std Dev (Decryption): {std_dec:.2f}", fontweight='bold',
        verticalalignment='bottom', horizontalalignment='center')
# 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
            plt.figure(figsize=(12, 8)) # Definir o tamanho da figura

            bar_width = 1
            index = range(1, len(times['encryption_time']) + 1) # Índice para o eixo X
            # Plotando o tempo de encriptação
            plt.plot([i - bar_width / 2 for i in index], times['encryption_time'], label=f'{folder_size} files')

            # Plotando o tempo de deciptação
            plt.plot([i - bar_width / 2 for i in index], times['decryption_time'], label=f'{folder_size} files')

            # Calculate the standard deviation for encryption and decryption
            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} files)")

            # Definir os ticks no eixo X para mostrar a cada 10 arquivos
            tick_positions = range(0, 101, 10) # Mostrar ticks a cada 10 arquivos
            plt.xticks(tick_positions)
            plt.legend()
            # Display both standard deviations below the title and above the plot

```

```
plt.text(0.25, 1.05, f"Std Dev (Encryption): {std_enc:.2f}"
        verticalalignment='bottom', horizontalalignment='center')

plt.text(0.75, 1.05, f"Std Dev (Decryption): {std_dec:.2f}"
        verticalalignment='bottom', horizontalalignment='center')
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 dada a média de tempo de encriptação e decriptação para os tamanhos 2, 4, 8, 16, 32, 64 e 128 bytes. Uma função com código da 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])
            avg_dec = sum(results[s]['decryption_time']) / len(results[s])
            # Calculate the standard deviation for encryption and decryption
            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')
        plt.plot(sizes, avg_dec_times, marker='x', color='red', label='Decryption')

        # 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, usando um line chart.

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

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

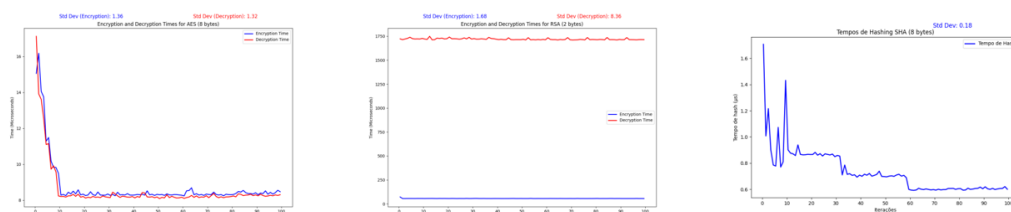
```
plt.plot(x_values, y_values, '-', linewidth=2, color='blue', label='Tempo médio de hash')
plt.xscale('log')
plt.xlabel('Tamanho do arquivo (bytes)', fontsize=12)
plt.ylabel('Tempo médio de hash (μs)', fontsize=12)
plt.title('Desempenho do SHA-256 por Tamanho de Arquivo', fontstyle='italic', fontsize=12)
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 resultados 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 menos credíveis.

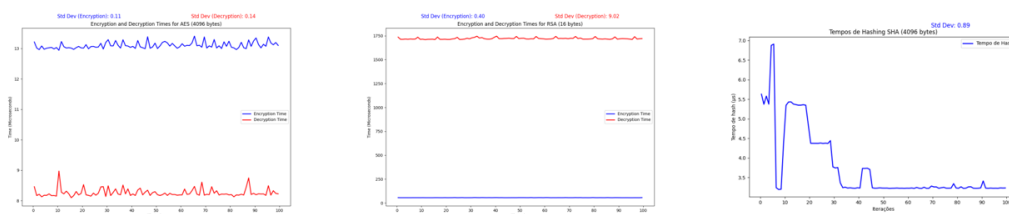
#### Ficheiros pequenos: comparação entre os resultados dos 3 modos



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 decifrou-os em 1750 μs, o que demonstra que é mais rápido encriptar um ficheiro do que descriptá-lo. Os desvios padrão coincidem

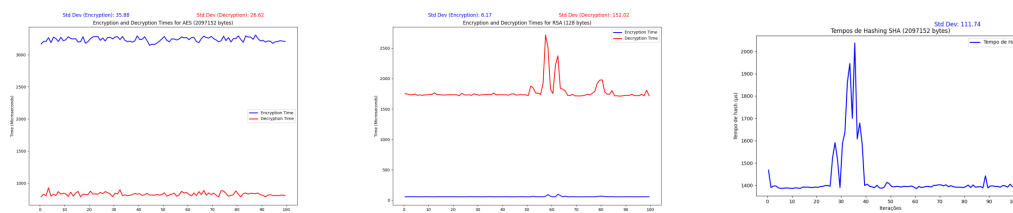
com isso, sendo de  $1.68 \mu s$  e  $8.36 \mu s$ , respetivamente. Por fim, no processo de Hashing (SHA-256), é possível ver uma ligeira descida de tempo, estagnando aproximadamente nos  $0.6 \mu s$ . O seu desvio padrão é igualmente pequeno, com  $0.18 \mu s$ . Permite-nos assim concluir que para ficheiros pequenos, o modo SHA é bastante rápido. No geral, o modo com tempos mais baixos, e consequentemente melhores, é o SHA-256. 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  $1.6 \mu s$ , e desce até aos  $0.6 \mu s$ . Conclui-se assim que, para ficheiros pequenos, a variação é quase nula.

## Ficheiros médios: comparação entre os resultados dos 3 modos



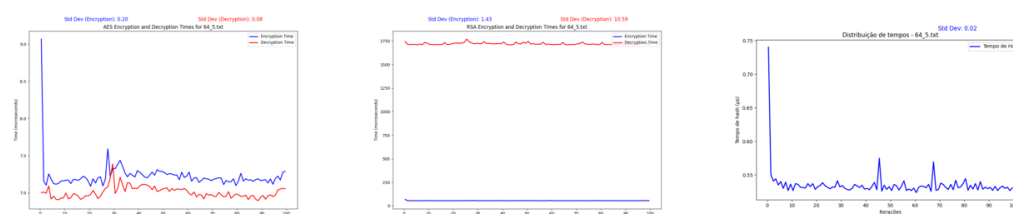
No que diz respeito a ficheiros médios, foram escolhidos para análise: 16 bytes para o RSA e 4096 bytes para os restantes modos. Com AES, o tempo de codificação, que ronda os  $13.1 \mu s$ , é maior que o tempo de descodificação, aproximadamente  $8.5 \mu s$ , sendo ambos estáveis e com um desvio padrão 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  $100 \mu s$  e decifra em  $1750 \mu s$ . Assim, os desvios padrão respetivos coincidem com as conclusões anteriores, com  $0.40 \mu s$  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  $7 \mu s$ , a partir do 50º ficheiro o tempo de hashing é de aproximadamente  $2 \mu s$ . O desvio padrão dos resultados obtidos é baixo, com um valor de  $0.89 \mu s$ . 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  $\mu$ s, e o tempo de descriptação é de mais de 500  $\mu$ s, havendo assim uma grande diferença entre tempos, o que indica que os processos envolvidos na encriptação podem ser mais exigentes que na descriptação. 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  $\mu$ s, e por isso o seu desvio padrão é alto, com 152.02  $\mu$ s. O desvio padrão de codificação continua baixo, com 6.17  $\mu$ s. 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  $\mu$ s. O desvio padrão reflete nessa variação, com o valor de 111.74  $\mu$ s. De resto, os tempos são baixos e estáveis, de aproximadamente 1400  $\mu$ s. 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

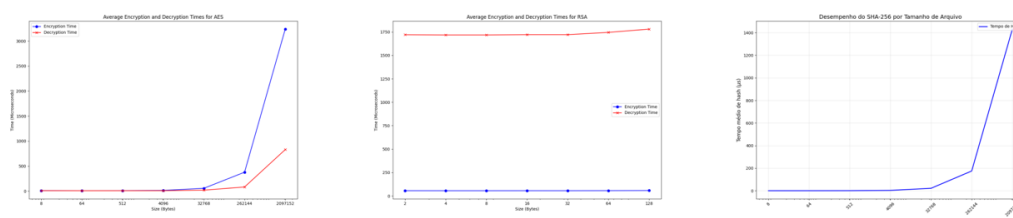


Para a análise das variações de tempo num ficheiro único, iterado 100 vezes, foi escolhido o ficheiro 64\_5.txt, de tamanho 64 bytes. Com o modo AES, a variação é quase nula, com tempos de codificação e descodificação que se encontram entre 6.8  $\mu$ s e 7.6  $\mu$ s. Isso reflete-se no desvio padrão, de valores 0.20 e 0.08 respetivamente. Usando o RSA, as variações são pequenas, situando-se principalmente na descriptação. O tempo de encriptação é de 100  $\mu$ s, como nos restantes tamanhos, e o tempo de descriptação é de



1750  $\mu$ s. Os desvios padrão são 1.43  $\mu$ s e 10.59  $\mu$ s, respetivamente. Como nos modos anteriores, o modo SHA não possui grandes variações, com tempos situados entre 0.53  $\mu$ s e 0.58  $\mu$ s, com um desvio padrão de 0.02  $\mu$ s. É, então, o modo com menos variações.

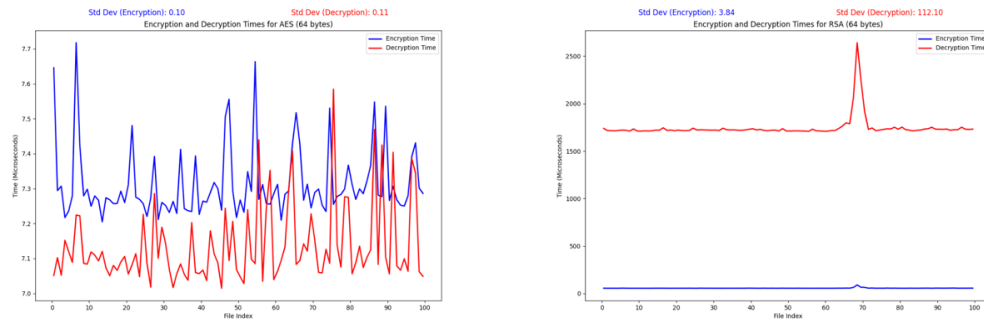
## Média de tempos: comparação entre os resultados dos 3 modos



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 descriptação. Pode-se concluir que o tamanho dos arquivos influencia, de certa forma, os tempos de encriptação e descriptação. O valor mais baixo é muito próximo de 0  $\mu$ s em ambos os processos. O valor mais alto é de mais de 3000  $\mu$ s para a encriptação, e de quase 1000  $\mu$ s para a descriptação. 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  $\mu$ s. Na decodificação ronda os 1750  $\mu$ s. É possível concluir que, para os tamanhos estudados, estes não influenciam o tempo de encriptação e deciptação, 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  $\mu$ s com o tamanho máximo de 2097152 bytes. O tempo mais baixo obtido foi de quase 0  $\mu$ s. Conclui-se assim que o tamanho de um ficheiro influencia o tempo de hashing.

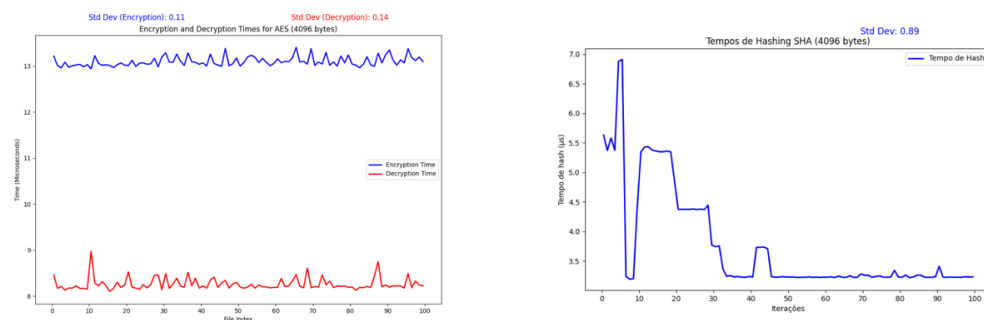
## Comparação entre encriptação AES e encriptação RSA





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  $\mu$ s e os 7.7  $\mu$ s, com um desvio padrão de 0.10  $\mu$ s. Já no RSA, ronda os 100  $\mu$ s, com um desvio padrão de 3.84  $\mu$ s. 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.

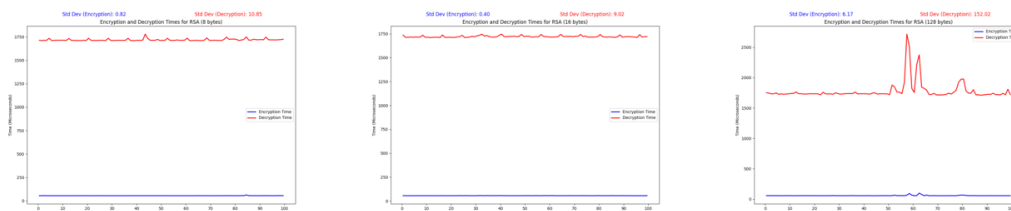
## Comparação entre encriptação AES e digest generation SHA-256



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 decodificação. 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  $\mu$ s, com um desvio padrão de 0.11  $\mu$ s. Já no modo SHA-256, ignorando as variações iniciais das primeiras 30 iterações, tem um tempo de 3  $\mu$ s, e um desvio padrão de 0.89  $\mu$ s. 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  $\mu$ s. 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 desenscriptação é mais lenta que a encriptação. Isto acontece porque tanto a encriptação como a desenscriptação 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 desenscriptação é quase tão grande como o próprio módulo. Por isso, duplicar o tamanho do módulo irá duplicar o tempo de encriptação e irá quadruplicar o tempo de desenscriptação. Uma solução em alguns casos é recorrer ao uso do Teorema Chinês do Resto (Chinese Remainder Theorem), que acelera a desenscriptação 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 decodificação, a diferença é grande. Nesses ficheiros, o tempo de codificação é de 100  $\mu$ s, com um desvio padrão de 1.68  $\mu$ s, e o tempo de decodificação é de 1750  $\mu$ s,

com um desvio padrão de 8.36  $\mu$ s, 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 desenhno altamente linear, mantendo

tempos de encriptação e deciptação significativamente pequenos, mesmo em tamanhos grandes. SHA... 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.