University of Porto

Department of Computer Science

Performance Benchmarking of Cryptographic Mechanisms Report

Alexandre Furriel
Henrique Teixeira
João Ferreira

A report submitted in partial fulfillment of the requirements of
the University of Porto for the degree of
Artificial Intelligence and Data Science in *Security and Privacy*

April 5, 2025

Contents

Chapter 1

Introduction

In a world where the Internet is part of our daily lives, cryptography plays a crucial role in maintaining digital communication and sensitive data secure as well as ensuring information integrity. Cryptography techniques can be classified into three main categories: message digests, which provide data integrity through the use of hash functions, symmetric cryptography, which ensures confidentiality using a shared key, and asymmetric cryptography, which enables secure key exchanges and digital signatures. In this report, we will dive into the performance measures of these techniques.

1.1 Problem Statement

Despite their widespread use, the performance characteristics of cryptographic algorithms vary significantly based on factors such as key length, processing power, and encryption/decryption speed. This report aims to address the question: How do message digests, symmetric cryptography, and asymmetric cryptography compare in terms of computational efficiency?

1.2 Aim and Objectives

Aim: The primary aim of this study is to analyze and compare the performance of message digests, symmetric cryptography, and asymmetric cryptography, highlighting their strengths and limitations.

Objectives:

- Evaluate computational efficiency of different cryptographic techniques.
- Evaluate if computational efficiency of algorithms differ when ran on the same file.
- Compare AES encryption and RSA encryption.
- Compare AES encryption and SHA digest generation.
- Compare between RSA encryption and decryption times.

1.3 Solution Approach

To achieve these objectives, we employ a methodology which involves implementing and benchmarking cryptographic algorithms on randomly generated files of different sizes. Performance measurements will be taken on the basis of execution time.

Chapter 2

Literature Review

Understanding cryptographic performance is crucial as security demands grow while computational efficiency remains critical. This chapter reviews existing research on benchmarking cryptographic algorithms and highlights gaps that this project addresses.

2.1 Current Research in Cryptographic Performance

Several studies have established frameworks to evaluate the speed and efficiency of cryptographic algorithms. Symmetric encryption, such as AES, is influenced by key size, block cipher modes, and hardware acceleration. Asymmetric algorithms, like RSA, are computationally intensive, especially during decryption. Cryptographic hash functions, such as SHA-256, show performance variations based on input size and implementation.

Many studies analyze cryptographic performance across different environments, however, fewer focus on software-based implementations using Python. This report builds on previous research by benchmarking AES, RSA, and SHA specifically in Python, using statistical validation.

2.2 Gaps in Existing Research

Despite extensive research, some gaps remain:

1. **Software-focused analysis**: Most studies prioritize hardware-optimized implementations, limiting relevance for software applications.
2. **Comprehensive comparison**: Research often examines only one cryptographic mechanism, making direct comparisons difficult.

This project addresses these gaps by systematically comparing AES, RSA, and SHA within a unified framework, emphasizing real-world applicability and statistical accuracy.

Chapter 3

Methodology

This chapter describes the methodology used to benchmark the performance of various cryptographic mechanisms: AES, RSA, and SHA. As indicated in Chapter 2, understanding the performance characteristics of these cryptographic operations is essential for making informed implementation decisions.

3.1 Algorithms Description

This section provides a detailed description of the three cryptographic mechanisms being evaluated.

3.1.1 AES (Advanced Encryption Standard)

AES (Advanced Encryption Standard) is a symmetric block cipher that operates on 128-bit blocks of data. It supports key sizes of 128, 192, and 256 bits, with AES-256 offering the highest level of security. The encryption process consists of 14 rounds, each involving SubBytes, ShiftRows, MixColumns, and AddRoundKey operations. Decryption is performed using the inverse of these operations. AES can be used in different modes of operation, such as ECB, CBC, etc., which define how blocks are processed in relation to each other. In this case, we used CBC mode with a key size of 256 bits.

3.1.2 RSA (Rivest-Shamir-Adleman)

The RSA algorithm is an asymmetric encryption algorithm widely used for secure data transmission. It relies on the mathematical difficulty of factoring large prime numbers. In RSA, two keys are used: a public key for encryption and a private key for decryption. The key size for RSA in this assignment is 2048 bits, which offers a strong level of security. The encryption operation involves raising the plaintext to the power of the public exponent and taking the remainder when divided by a large modulus (RSA-encryption = $\text{plaintext}^e \mod n$). The decryption operation involves raising the ciphertext to the power of the private exponent and taking the remainder when divided by the same modulus (RSA-decryption = $\text{ciphertext}^d \mod n$). The mathematical foundation of RSA is based on number theory, specifically the use of modular exponentiation and the difficulty of factoring large composite numbers, which ensures the security of the algorithm.

3.1.3 SHA-256 (Secure Hash Algorithm)

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that produces a fixed-size output of 256 bits. It is a one-way hash function, meaning it is computationally infeasible to reverse the process and obtain the original input from the hash. The algorithm processes input data in blocks, applying a series of mathematical operations including bitwise operations, modular additions, and message expansions, to produce a unique hash value for each unique input. SHA-256 is widely used in various security protocols and applications due to its resistance to collision attacks and its fixed output size of 256 bits.

3.2 Implementation

3.2.1 Development Environment

The development environment consists of the following components:

- **Python Version**: 3.10.2, running in a Jupyter Notebook environment.
- **Operating System**: Linux Ubuntu.
- **Hardware Specifications**:
  - **CPU**: AMD Ryzen 7 7840HS, 8 cores, 16 threads, base clock 3.8 GHz, boost up to 5.137 GHz, 16 GiB RAM.
- **Relevant Software Dependencies**: NumPy, Matplotlib, Jupyter Notebook, Git.

3.2.2 Python Cryptography Libraries

The implementation uses the Python cryptography library with its hazmat primitives as specified in the assignment.

```
# Example imports - replace with your actual implementation
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
    modes
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
import os
```

Listing 3.1: Example import statements for cryptography implementation

3.2.3 Test Data Generation

For the full implementation in Python, refer to Appendix .1.

Random text files were generated with the following sizes:

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

3.3 AES Implementation and Testing

Algorithm 1 AES Encryption and Decryption Performance Test

Input: Directory D, File size s, number of iterations t
Output: Encryption and decryption performance metrics: mean, variance, and standard error

function TestAES(D, s, t, r)
    Initialize result lists $R_{\text{enc}}, R_{\text{dec}} \leftarrow \emptyset$
    Initialize F with directory D and the filename given by size s
    for $i = 0$ to $t - 1$ **do**
        Generate random 32-byte key k
        Generate random 16-byte IV iv
        regenerate file F
        Read plaintext p from file f
        Pad p to be a multiple of 16 bytes

        Measure encryption times over t iterations, repeated r times
        Store encryption times in R_{enc}
        Encrypt p using AES-CBC with key k and IV iv, producing ciphertext c

        Measure decryption times over t iterations, repeated r times in microseconds
        Store decryption times in R_{dec}
    end for
    Compute mean, variance, and standard error for R_{enc} and R_{dec}
    return $(\text{mean}_{\text{enc}}, \text{var}_{\text{enc}}, \text{stderr}_{\text{enc}}), (\text{mean}_{\text{dec}}, \text{var}_{\text{dec}}, \text{stderr}_{\text{dec}})$
end function

3.4 SHA-256 Implementation and Testing

Algorithm 2 SHA-256 Hashing Performance Test

Input: Directory D, File size s, number of iterations t
Output: Performance metrics: mean, variance, and standard error of hashing time

function ComputeSHA256(f)
    Initialize SHA-256 digest H
    Read file f into memory as D
    Start timer t_s
    Update digest H with data D
    Finalize digest to obtain hash h
    Stop timer t_e
    Compute elapsed time $\Delta t = (t_e - t_s) \times 10^6$ (microseconds)
    return $(\Delta t, h)$
end function

function FileTime(f)
    Extract and return Δt from ComputeSHA256(f)
end function

function TestSHA256(D, s, t)
    Initialize result list $R \leftarrow \emptyset$
    Initialize F with directory D and the filename given by size s
    for $i = 0$ to $t - 1$ **do**
        regenerate file F
        Append FileTime(f) to R
    end for
    Compute mean, variance, and standard error of R
    return $(\text{mean}(R), \text{var}(R), \text{stderr}(R))$
end function

3.5 RSA Implementation and Testing

Algorithm 3 RSA Key Generation and Encryption/Decryption

function GenerateKeys
    Generate private key with RSA (65537, 2048 bits) $\rightarrow$ `private_key`
    Derive public key from private key $\rightarrow$ `public_key`
    return `private_key`, `public_key`
end function

function EncryptRSA(plaintext, public_key)
    Start timer t_s
    Encrypt plaintext using RSA-OAEP with SHA-256 $\rightarrow$ ciphertext
    Stop timer t_e
    Compute encryption time $\Delta t = (t_e - t_s) \times 10^6$ (microseconds)
    return $(\Delta t, \texttt{ciphertext})$
end function

function DecryptRSA(ciphertext, private_key)
    Start timer t_s
    Decrypt ciphertext using RSA-OAEP with SHA-256 $\rightarrow$ plaintext
    Stop timer t_e
    Compute decryption time $\Delta t = (t_e - t_s) \times 10^6$ (microseconds)
    return Δt
end function

function RSAEncryptDecrypt(plaintext)
    $(\Delta t_{enc}, \texttt{ciphertext}) \leftarrow$ EncryptRSA(plaintext, public_key)
    $\Delta t_{dec} \leftarrow$ DecryptRSA(ciphertext, private_key)
    return $(\Delta t_{enc}, \Delta t_{dec})$
end function

Algorithm 4 Encrypt and decrypt the same file 1000x for each size

```
1: function GetSameFileInfo
2:     Initialize dictionary D to store times per file size
3:     for all files f in rsa_directory do
4:         if f is a regular file then
5:             Read plaintext P from f
6:             Get file size s
7:             for i = 1 to 1000 do
8:                 (t_e, t_d) ← RSAEncryptDecrypt(P)
9:                 Append t_e to D[s].encrypt_times
10:                Append t_d to D[s].decrypt_times
11:            end for
12:        end if
13:    end for
14:    return D
15: end function
```

Algorithm 5 Encrypt and decrypt 1000x randomly generated files of each size

```
function GetRandFileInfo
    Initialize dictionary D to store times per file size
    for i = 1 to 1000 do
        Generate random RSA files
        for all files f in rsa_directory do
            if f is a regular file then
                Read plaintext P from f
                Get file size s
                (t_e, t_d) ← RSAEncryptDecrypt(P)
                Append t_e to D[s].encrypt_times
                Append t_d to D[s].decrypt_times
            end if
        end for
    end for
    return D
end function
```

3.6 Data Analysis Methodology

To evaluate the encryption and decryption performance of the RSA algorithm, a statistical analysis was conducted based on two distinct scenarios:

- **Scenario 1: Fixed File**
  A set of files with identical content was used to measure time variation introduced solely by the randomness of the algorithm and the system. A single file was processed **1000 times**, recording the encryption and decryption time for each execution. This method allowed for obtaining robust performance metrics for a single file.
- **Scenario 2: Random Files**
  A total of **1000 different files** were generated and processed, each containing random content. Each file was encrypted and decrypted once, allowing the observation of execution time variation with different file sizes and contents.

In both scenarios, the collected data included the encryption and decryption times for each file. From this data, the following descriptive statistics were computed:

- **Mean (μ)**: The average execution time.
- **Variance (σ^2)**: A measure of the dispersion of the times around the mean.
- **Standard Error of the Mean (SEM)**: An estimate of the uncertainty of the mean, calculated as:
  $$\text{SEM} = \frac{\sigma}{\sqrt{n}}$$
  where σ represents the standard deviation of the times and n the number of samples (1000).
- **Confidence Interval (CI)**: The confidence interval for the mean was calculated to assess the range in which the true mean lies, using the formula:
  $$CI = \mu \pm Z \times \text{SEM}$$
  where μ is the mean, Z is the critical value corresponding to the desired confidence level (e.g. for 95 percent confidence, $Z \approx 1.96$), and SEM is the standard error of the mean.)

This process was applied separately for encryption and decryption times. The obtained results were then used to generate graphs that analyze the behavior and efficiency of the RSA algorithm under different conditions.

Chapter 4

Results

This chapter presents the results obtained from the implementation and execution of the proposed methodology. The findings are structured logically and include relevant performance metrics, data analysis, and system evaluation.

4.1 Overview of Results

The following figures provide a visual summary of algorithm performance. AES shows consistently fast encryption/decryption times across file sizes. RSA, while consistent across all test cases, seems slower than AES, although we only have a 2 test case comparison. SHA-256 scales predictably with input size, showing steady increases in digest time.

4.2 Performance Evaluation

The experiments were conducted following a structured methodology where multiple file sizes were tested, ensuring statistically significant results. The tests were performed on a system with the following specifications:

- Processor: AMD Ryzen 7 7840HS, 8 cores, 16 threads, base clock 3.8 GHz, boost up to 5.137 GHz
- RAM: 16 GB DDR4
- Operating System: Ubuntu 20.04
- Python Version: 3.10.2
- Cryptography Library: Python cryptography module (hazmat primitives)

File Size (bytes)	Encryption		Decryption	
	Mean (µs)	± CI (µs)	Mean (µs)	± CI (µs)
8	3.645e+01	2.659e-04	2.322e+01	2.681e-05
64	4.288e+01	3.890e-04	2.636e+01	1.311e-04
512	4.003e+01	1.367e-04	2.448e+01	3.012e-05
4096	4.685e+01	1.168e-04	2.528e+01	3.754e-05
32768	8.819e+01	1.197e-03	3.412e+01	1.778e-04
262144	3.367e+02	3.172e-03	8.507e+01	3.767e-04
2097152	2.733e+03	2.720e-02	1.008e+03	1.079e-01

Table 4.1: AES Encryption and Decryption Results (95% Confidence Intervals)

Test #	Encryption		Decryption	
	Mean (µs)	± CI (µs)	Mean (µs)	± CI (µs)
1	3.263e+01	5.547e+01	2.341e+02	3.361e+03
2	3.296e+01	4.407e+02	2.353e+02	6.275e+03
3	3.309e+01	3.628e+02	2.348e+02	5.277e+03
4	3.336e+01	4.022e+02	2.405e+02	1.639e+04
5	3.341e+01	9.116e+02	2.382e+02	9.403e+03
6	3.487e+01	3.304e+02	2.386e+02	9.877e+03
7	5.227e+01	7.319e+02	2.480e+02	7.768e+03

Table 4.2: RSA Encryption and Decryption Results (95% Confidence Intervals)

File Size (Bytes)	Mean (µs)	± CI (µs)
8	6.048e+00	1.199e+01
64	6.124e+00	1.278e+01
512	6.255e+00	7.216e+00
4096	8.114e+00	1.022e+01
32768	2.729e+01	1.353e+02
262144	1.425e+02	1.841e+02
2097152	1.038e+03	3.708e+04

Table 4.3: SHA-256 Performance Results (95% Confidence Intervals)

4.3 Visual Representation of Results

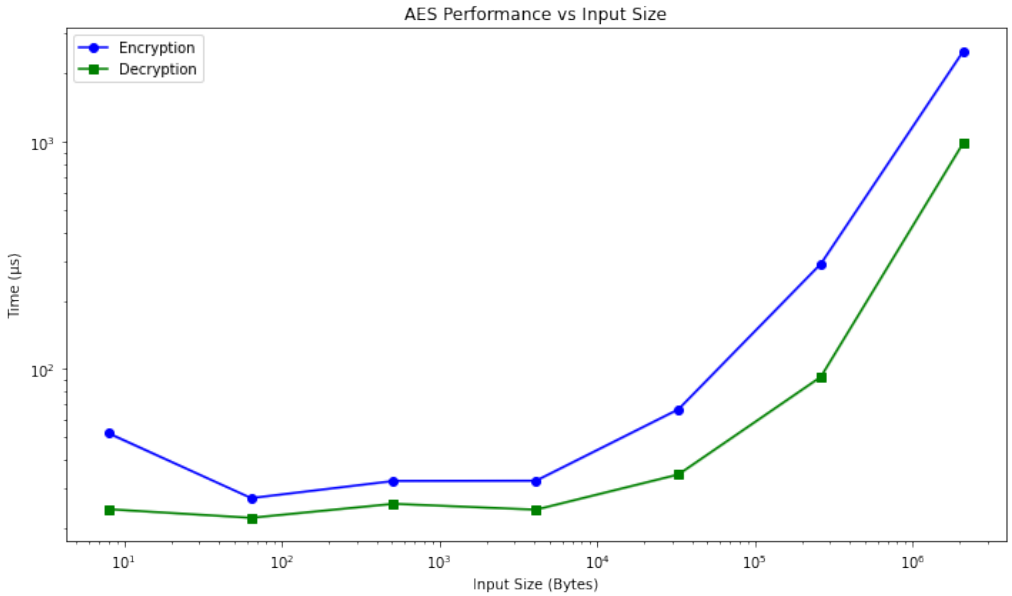

Figure 4.1: AES Performance on the Same File

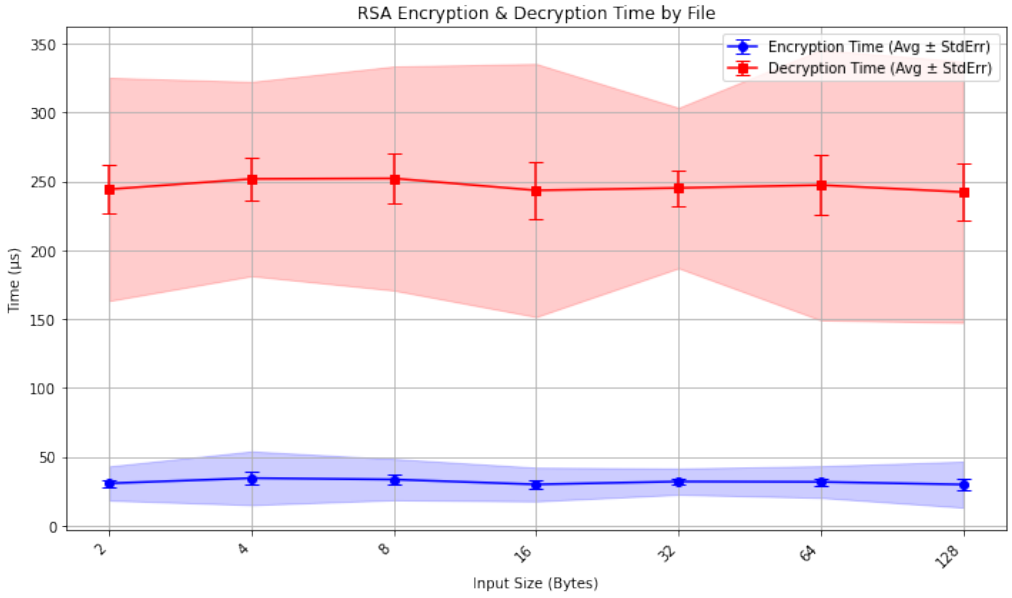

Figure 4.2: RSA Performance on the Same File

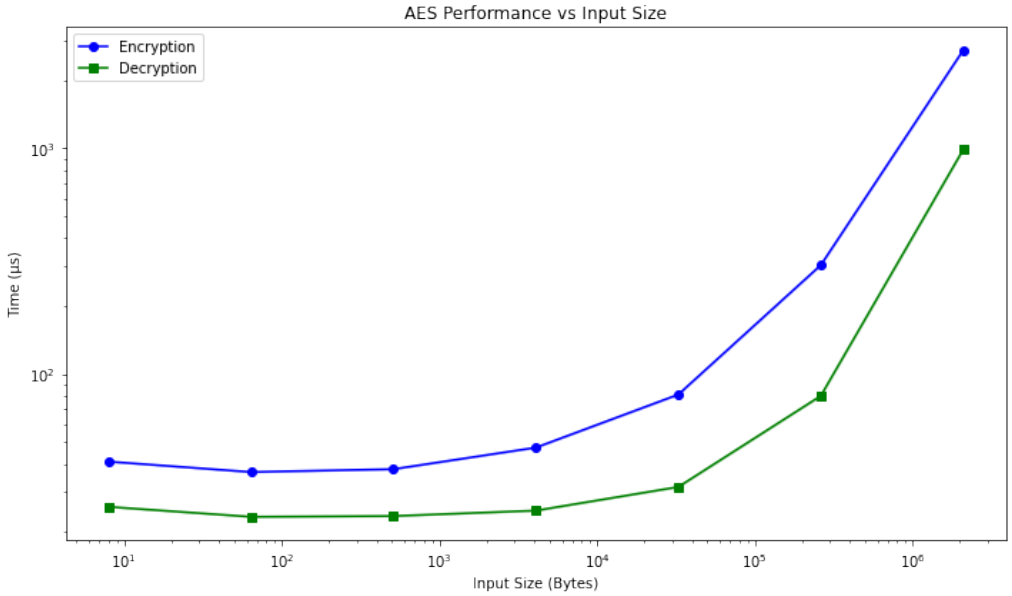

Figure 4.3: AES Performance on the Different Files

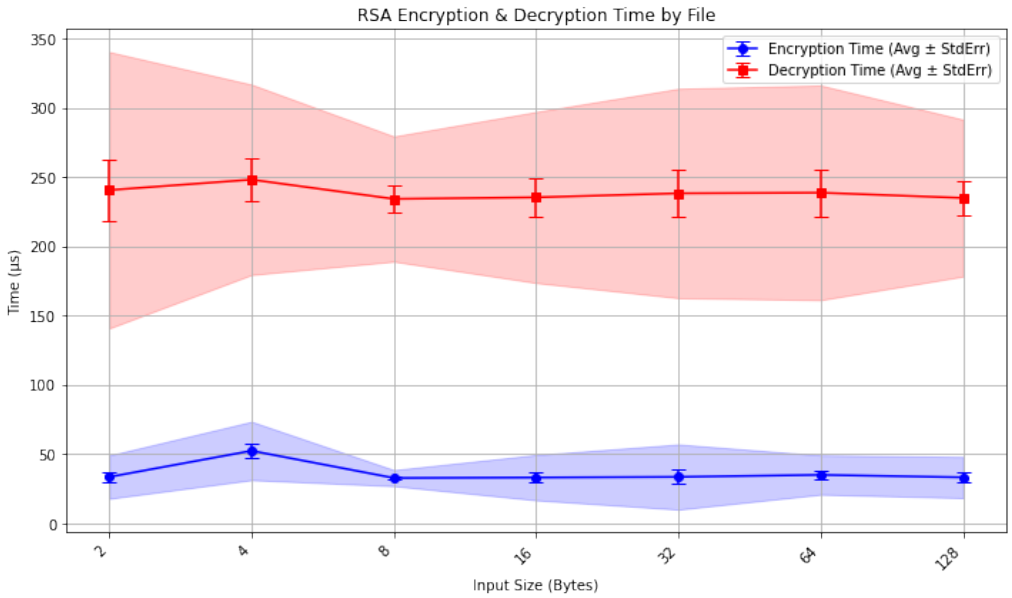

Figure 4.4: RSA Performance on the Different Files

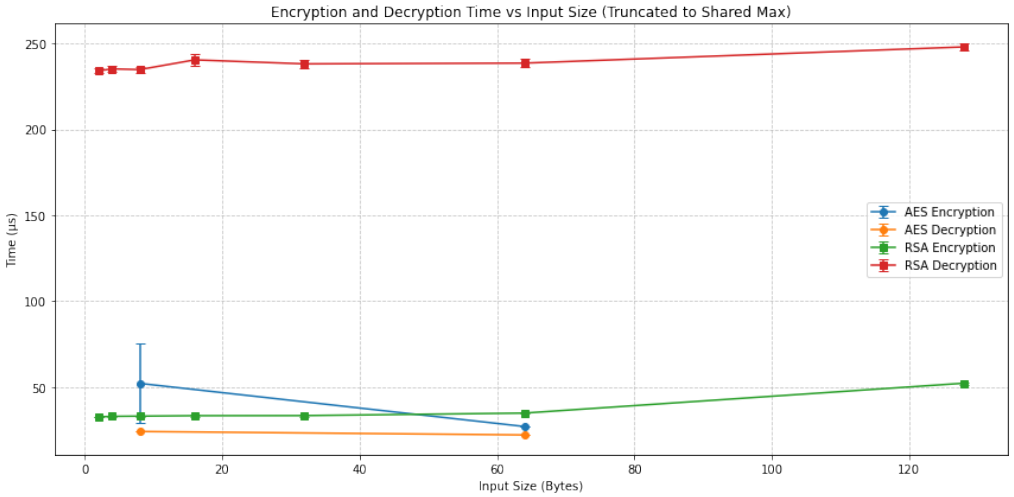

Figure 4.5: RSA vs AES Performance on the Different Files

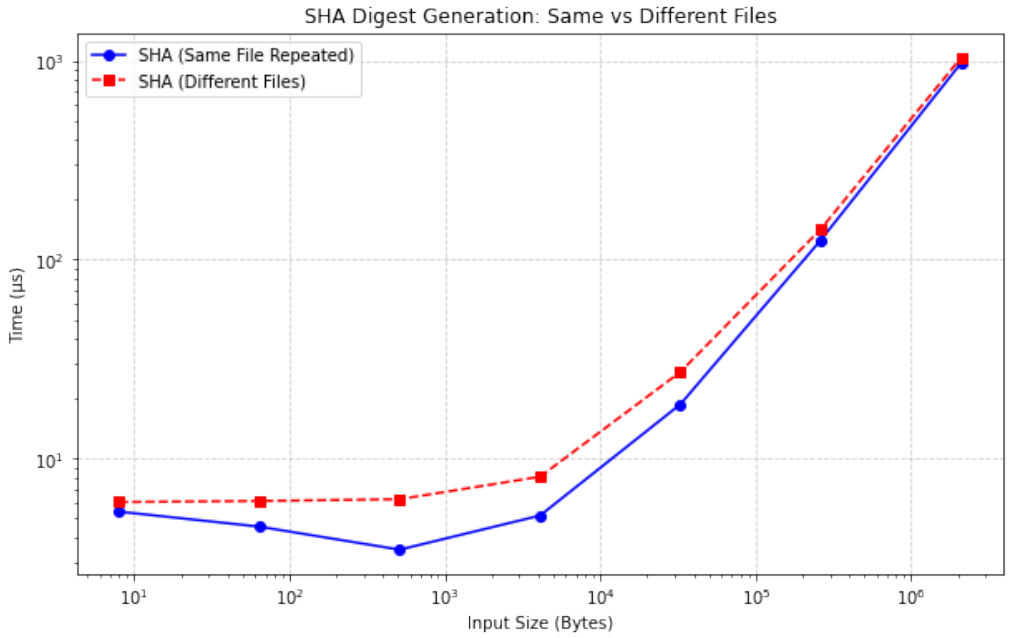

Figure 4.6: SHA-256 Performance

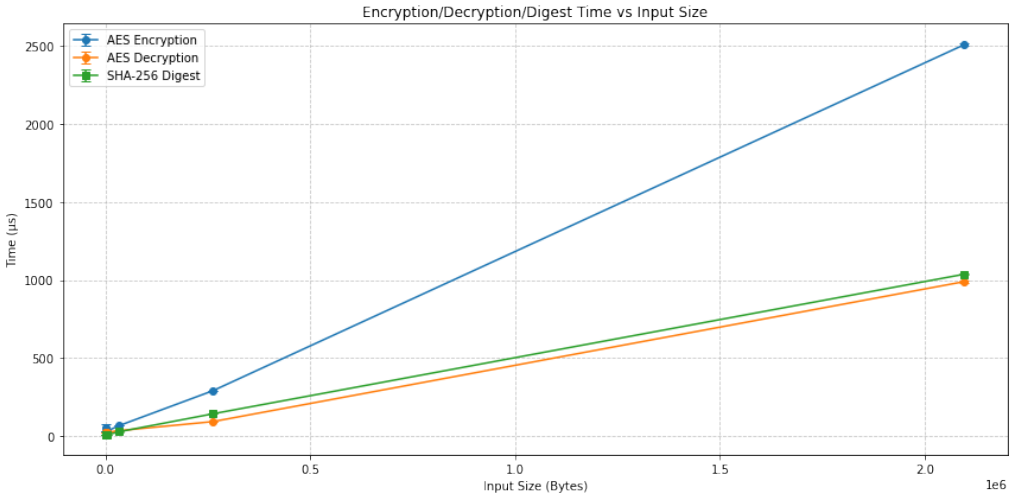

Figure 4.7: AES Encryption/Decryption vs SHA-256 Digest Performance

Chapter 5

Discussion and Analysis

This chapter provides an interpretation of the results presented in Chapter 4. It discusses the significance of the findings, evaluates the performance of the implemented approach, and addresses limitations.

5.1 Analysis of Results

In the results shown in the previous chapter, we notice that running the same algorithm over the same file multiple times does not change the duration of the runtime in a significant statistical way (Figure 4.1 and Figure 4.2). However, we can see the results can vary in comparison to the mean of multiple randomly generated files, which suggests AES takes slightly more or less time depending on the content of the given file. This is to be expected, because different inputs to the algorithm can have different complexity of encryption/decryption.
AES took significantly less time to process files compared to RSA in decryption, while the encryption time was essentially the same, as seen in Figure 4.5. However, we do not have a sufficient number of test cases with different file sizes to conclude the expected tendency of the results, given that we could only compare the files of size 8 and 64 bytes. This was expected in any case, because AES is a symmetric cryptographic algorithm while RSA is asymmetric. AES takes less time to decrypt the data because it uses the same key used in encryption, resulting in a lower mathematical complexity in the decryption, which in RSA does not happen, as it uses a public-private key pair for encryption-decryption of messages.
We also noticed while benchmarking RSA, with multiple number of test cases from 100 to 10 000, the confidence interval length is consistently about 10x larger than the mean value (Table 4.2). This makes us conclude that the encryption and decryption times vary significantly depending on the content of the file.
The decryption mean time of RSA is on average 10x slower then its encryption mean time on random generated files, probably due to the mathematical complexity of the algorithm behind decryption as is expected.
SHA-256, as a hashing algorithm, consistently demonstrated the faster processing times compared to AES as seen in Figure 4.7. However, since hashing serves a different purpose than encryption, its role in data security is fundamentally distinct from that of AES and RSA encryption and decryption.
In terms of scalability, both AES and SHA-256 appear constant for small file sizes and then to scale linearly for larges ones (Figure 4.3 and Figure 4.6). Regarding RSA, its time complexity is relatively constant for small sizes as well (Figure 4.4).

5.2 Limitations

- Due to the small file sizes for testing RSA, we were unable to see any significant time changes between smaller and larger files, leading to an approximately constant encrypting and decrypting time.
- Possible sources of error or variability in the results.
- Hardware Variability: Tests were performed on a specific CPU architecture; results may vary significantly on different hardware, particularly when comparing performance on devices with different levels of cryptographic hardware acceleration.
- Aspects that could be improved or optimized in future work.

Chapter 6

Conclusion

6.1 Summary of the Study

This study conducted a comprehensive performance benchmarking of three fundamental cryptographic mechanisms: AES (symmetric encryption), RSA (asymmetric encryption), and SHA-256 (hashing). Using a Python implementation with the cryptography library's hazmat primitives, we measured execution times across various file sizes, running multiple iterations to ensure statistical significance. The primary objective was to quantify the computational costs of these operations to inform design decisions for secure systems where performance considerations are important.

.1 Implementation Code

AES Testing

```
def test_aes(directory: str, size: int, amount: int = 100, same_file: bool = False) -> tuple[tuple[float, float, float], tuple[float, float, float]]:
    assert amount > 0

    results: list[list[float]] = [[], []]
    filename: str = f"random_text_{size}.bin"

    for i in range(amount):
        key: bytes = os.urandom(32)
        iv: bytes = secrets.token_bytes(16)

        if not same_file or i < 1:
            write_to_file(directory, filename, size)

        path: str = os.path.join(directory, filename)
        with open(path, "rb") as plaintext_file:
            plaintext = plaintext_file.read()

        padding_length = 16 - len(plaintext) % 16
        padded_plaintext = plaintext + bytes([padding_length]) * padding_length

        encrypt_trials = array(timeit.repeat(
            lambda: Cipher(algorithms.AES(key), modes.CBC(iv)).encryptor().update(padded_plaintext) +
                    Cipher(algorithms.AES(key), modes.CBC(iv)).encryptor().finalize(),
            number = 1,
            repeat = 1
        ))

        ct: bytes = Cipher(algorithms.AES(key), modes.CBC(iv)).encryptor().update(padded_plaintext) + \
                    Cipher(algorithms.AES(key), modes.CBC(iv)).encryptor().finalize()

        decrypt_trials = array(timeit.repeat(
            lambda: Cipher(algorithms.AES(key), modes.CBC(iv)).decryptor().update(ct) +
                    Cipher(algorithms.AES(key), modes.CBC(iv)).decryptor().finalize(),
            number = 1,
            repeat = 1
        ))

        results[0].extend(encrypt_trials)
        results[1].extend(decrypt_trials)

    mean_encrypt = mean(results[0]) * 1e6
    mean_decrypt = mean(results[1]) * 1e6

    var_encrypt = var(results[0], ddof = 1) * 1e6
    var_decrypt = var(results[1], ddof = 1) * 1e6

    stderr_encrypt = std(results[0], ddof = 1) / sqrt(len(results[0])) * 1e6
```

```
    stderr_decrypt = std(results[1], ddof = 1) / sqrt(len(results[1])) * 1e6

    return (
        (mean_encrypt, var_encrypt, stderr_encrypt),
        (mean_decrypt, var_decrypt, stderr_decrypt)
    )
```

SHA-256 Testing

```
def compute_sha256(file_path: str) -> tuple[float, bytes]:
    digest = hashes.Hash(hashes.SHA256())

    with open(file_path, 'rb') as file:
        data = file.read()

    start = timer()
    digest.update(data)
    hashing = digest.finalize()
    end = timer()

    elapsed_time_us = (end - start) * 1000000
    return elapsed_time_us, hashing

def file_time(filename: str) -> float:
    time, _ = compute_sha256(filename)
    return time

def print_results(test: str, result: tuple[float, float, float]) -> None:
    print(f"Test {test}:")
    print(f"\t\tmean: {result[0][0]:.5f}\n\tvariance: {result[0][1]:.5e}\n\std error: {result[0][2]:.5e}")

def test_sha256(directory: str, size: int, amount: int = 100, same_file: bool = False) -> tuple[float, float, float]:
    assert amount > 0

    results = []
    filename: str = f"random_text_{size}.bin"

    for i in range(amount):
        if not same_file or i < 1:
            write_to_file(directory, filename, size)
        path: str = os.path.join(directory, filename)
        results.append(file_time(path))

    return (mean(results), var(results, ddof = 1), std(results, ddof = 1) / sqrt(amount))
```

RSA Testing

```
def rsa_encryption(plaintext, public_key):
    start = timer()
    ciphertext = public_key.encrypt(
        plaintext,
        padding.OAEP(
            mgf = padding.MGF1(algorithm=hashes.SHA256()),
            algorithm = hashes.SHA256(),
            label = None
        )
    )
    end = timer()
```

```
12
13      encryption_time: float = (end - start) * 1000000    # microseconds
14      return encryption_time, ciphertext
15
16  def rsa_decryption(ciphertext, private_key):
17      start = timer()
18      plaintext = private_key.decrypt(
19          ciphertext,
20          padding.OAEP(
21              mgf = padding.MGF1(algorithm=hashes.SHA256()),
22              algorithm = hashes.SHA256(),
23              label = None
24          )
25      )
26      end = timer()
27      decryption_time: float = (end-start) * 1000000 # microseconds
28
29      return decryption_time
30
31  def rsa_encrypt_decrypt(plaintext):
32      encrypt_time, ciphertext = rsa_encryption(plaintext, public_key)
33      decrypt_time = rsa_decryption(ciphertext, private_key)
34      return encrypt_time, decrypt_time
```