Performance Benchmarking of Cryptographic Mechanisms

Security and Privacy - ASSIGNMENT 1

1st João Guedes
Faculdade de Ciências
Universidade do Porto
Porto, Portugal
up202203859@up.pt

2nd Ricardo Figueiredo
Faculdade de Ciências
Universidade do Porto
Porto, Portugal
up202105430@up.pt

Abstract

Cryptographic mechanisms play a crucial role in securing digital communications by ensuring confidentiality, integrity, and authentication. This study benchmarks the performance of three widely used cryptographic algorithms: AES (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), and SHA-256 (Secure Hash Algorithm 256-bit). The evaluation is conducted by measuring encryption, decryption, and hashing times for files of varying sizes using a Python implementation. The results highlight the trade-offs between security and computational cost, providing insights into the practical deployment of these algorithms in real-world applications.

Index Terms

cryptography, benchmark, AES, RSA, SHA-256

I. Introduction

With the increasing reliance on digital security, cryptographic mechanisms have become essential for protecting sensitive information. Cryptographic techniques should be considered for the protection of data that is sensitive, has a high value, or is vulnerable to unauthorized disclosure or undetected modification during transmission or while in storage. [3]

This study focuses on measuring the execution time of three cryptographic mechanisms: AES (symmetric encryption), RSA (asymmetric encryption), and SHA-256 (hashing). Using Python implementations, we assess their performance by encrypting, decrypting, and hashing files of various sizes.

II. Methods

A benchmarking framework was implemented in **Python**, utilizing the `cryptography` library for AES and RSA operations, and the `hashes` module for SHA-256 computations. Time measurements were conducted using Python's `timeit` module to ensure accuracy and repeatability. [1,2,4] The following sections describe the methodology in detail.

A. Key Generation

It is important to note that the cryptographic keys used in the benchmarks were generated only once, prior to executing the tests. Specifically, a random 256-bit key for AES and a 2048-bit RSA key pair (public and private keys) for RSA operations were created once. The time required for key generation was not included in the benchmark measurements, ensuring that only the encryption, decryption, and hashing operations were timed.

B. File Generation

To ensure statistically significant results, multiple random files of varying sizes were generated for benchmarking AES, RSA, and SHA-256 cryptographic operations. [5] The `generate_files.py` script was responsible for creating these files:

- **AES/SHA-256 File Sizes:** 8B, 64B, 512B, 4KB, 32KB, 256KB, 2MB.
- **RSA File Sizes:** 2B, 4B, 8B, 16B, 32B, 64B, 128B.
- **Number of Files:** Each file size had five instances to allow averaging results.

C. AES Encryption and Decryption Benchmarking

The `aes_benchmark.py` script tested AES-256 encryption and decryption using the CBC mode with PKCS7 padding. A random 256-bit key was used for all tests.

1) Encryption Process:

1.1 Generate a random IV (16 bytes).

1.2 Apply PKCS7 padding to the input data.

1.3 Encrypt the padded data using AES in CBC mode.

1.4 Prepend the IV to the ciphertext for later decryption.

2) Decryption Process:

2.1 Extract the IV from the ciphertext.

2.2 Decrypt using AES-CBC.

2.3 Remove PKCS7 padding.

Each operation was timed using the `timeit` module over 100 iterations, and the results were averaged in microseconds (μs).

D. RSA Encryption and Decryption Benchmarking

The `rsa_benchmark.py` script performed RSA encryption and decryption using a 2048-bit RSA key with OAEP padding (SHA-256). Due to the inherent size limitations of RSA, which restrict encryption to small data blocks, all RSA operations were performed on entire files in a single block. This approach is valid because the file sizes chosen (ranging from 2B to 128B) are small enough to be encrypted directly using RSA, thereby avoiding the need for hybrid encryption.

1) Encryption Process:

1.1 Encrypt the plaintext file using the RSA public key.

2) Decryption Process:

2.1 Decrypt the ciphertext using the RSA private key.

Each operation was timed over 10 iterations (due to RSA's computational intensity) and averaged in microseconds (μs).

E. SHA-256 Hashing Benchmarking

The `sha_benchmark.py` script measured the time required to compute a SHA-256 digest for each file.

1) Process:

1.1 Create a new SHA-256 hash object per iteration.

1.2 Compute the hash for the entire file content.

Each operation was timed over 1000 iterations and averaged in microseconds (µs).

F. Benchmark Execution and Data Collection

The `benchmark_runner.py` script orchestrated the benchmarking process:

1) Iterate through generated files.
2) Execute AES encryption/decryption, RSA encryption/decryption, and SHA-256 hashing.
3) Generate analysis output.

To ensure consistency and minimize the impact of cold-start effects, caches were properly **warmed up** before collecting measurements. [6]

The benchmarks were executed on the following machine:

- **CPU:** AMD Ryzen 3 3100 4C/8T 3.6GHz
- **Motherboard:** MSI A320M-A PRO
- **RAM:** G.SKILL Ripjaws V 16GB 3200MT/S
- **Storage:** KingSpec SSD 512GB 3D NAND
- **OS:** Linux Mint 22.1 'Xia' - Cinnamon
- **Python version:** 3.12
- **Environment:** Network cable unplugged.

G. Data Visualization

The `plot_results.py` script processed the benchmark results and generated plots illustrating:

- AES encryption vs decryption time.
- RSA encryption vs decryption time.
- SHA-256 hash computation time.
- Comparative performance analysis of AES, RSA, and SHA-256.

Each graph was plotted using a logarithmic X-axis (file size in bytes) and a linear Y-axis (time in μs).

III. Results

A. Performance Analysis of Cryptographic Algorithms

Fig. 1 illustrates the performance of AES-256 encryption and decryption. As expected, execution time grows with file size. Both operations scale linearly, but encryption consistently takes slightly longer than decryption. Performance remains negligible for small files (under 4KB), while larger files (256KB and above) introduce notable delays, reaching several thousand microseconds.

Fig. 2 shows the time required to encrypt and decrypt small files using RSA-2048 with OAEP padding. Unlike AES, RSA performance is mostly independent of file size due to its design constraints—only small chunks can be encrypted at once. Decryption takes significantly longer than encryption, as is typical with RSA. This asymmetry confirms why RSA is rarely used for large data encryption, favoring hybrid approaches.

Fig. 3 presents the time required to compute SHA-256 hashes. The execution time increases steadily with file size, closely mirroring AES behavior. For small files (under 4KB), performance is virtually instant, but files above 256KB exhibit a marked rise in hashing time, as expected from a linear-time digest algorithm.

B. Impact of Padding on Encryption Times

Block ciphers like AES require the input data to be a multiple of the block size (16 bytes for AES). Consequently, when encrypting small files, padding (using PKCS7 in our implementation) can significantly inflate the effective data size. For example, an 8-byte file is padded to a full 16-byte block, doubling the amount of data processed. This increase is not due to the encryption algorithm itself, but rather due to the overhead incurred from padding, which adds extra computation for both padding and later unpadding operations. As a result, the encryption time for very small files may be disproportionately high compared to their original size. However, for larger files, the fixed overhead of padding becomes negligible relative to the total data size.

IV. Conclusions

This benchmarking study provided a comparative performance analysis of three widely used cryptographic algorithms: AES, RSA, and SHA-256. The results highlight clear distinctions in their computational characteristics.

AES-256 exhibited efficient and consistent performance across a wide range of file sizes, with encryption taking slightly longer than decryption. Its linear scalability and low overhead for small files reinforce its suitability for general-purpose symmetric encryption tasks.

RSA-2048, on the other hand, demonstrated significant asymmetry between encryption and decryption times. Due to its reliance on modular arithmetic, encryption remained relatively fast even for larger files, but decryption showed notable latency. Additionally, its performance remained largely independent of file size, reflecting its design for encrypting small data blocks—validating the common use of RSA in hybrid encryption schemes rather than bulk data encryption.

SHA-256 hashing performance scaled steadily with file size, with execution times closely following AES behavior. For small files, the hashing overhead was minimal, but larger files introduced measurable delays, as expected from its iterative compression function design.

Moreover, the performance outcomes can be directly linked to the underlying algorithm designs. AES's block cipher structure, which processes data in fixed-size blocks, naturally supports linear scaling as file sizes increase. In contrast, RSA's modular exponentiation, a computationally intensive operation, constrains its practicality to encrypting small amounts of data. This fundamental difference underscores the need for careful algorithm selection based on the specific requirements and constraints of the application.

Overall, the benchmarking results confirm theoretical expectations and practical usage patterns of these cryptographic algorithms. The study also emphasizes the importance of using appropriate cryptographic tools depending on data size and application context.

Acknowledgment

We would like to express our sincere gratitude to Professor Manuel Eduardo Correia for providing us with the insightful reading material that greatly contributed to the development of this work.

REFERENCES

[1] Nielson, S. J., & Monson, C. K. (2019). Hashing. In: Practical Cryptography in Python. Apress.

[2] Byrne, D. (2021). Asymmetric encryption. In: Full Stack Python Security: Cryptography, TLS, and Attack Resistance. Simon and Schuster.

[3] Barker, E. (2016). Introduction. In: Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms. NIST Special Publication 800-175B, 1-79.

[4] Python Software Foundation. (2024). *timeit — Measure execution time of small code snippets*. Python 3.12. Official Documentation.

[5] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., & Vo, S. (2010). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22 Rev. 1a.

[6] Intel Corporation. (2023). Volumes 3A and 3B. In: Intel 64 and IA-32 Architectures Optimization Reference Manual.

APPENDIX

PROJECT FILE OVERVIEW

- `aes_benchmark.py` – Benchmarks AES encryption and decryption.
- `rsa_benchmark.py` – Benchmarks RSA encryption and decryption.
- `sha_benchmark.py` – Benchmarks SHA-256 hashing.
- `generate_files.py` – Generates random files.
- `benchmark_runner.py` – Coordinates and executes all benchmarks.
- `plot_results.py` – Generates graphs from benchmark results.

PROJECT RESULTS PLOT

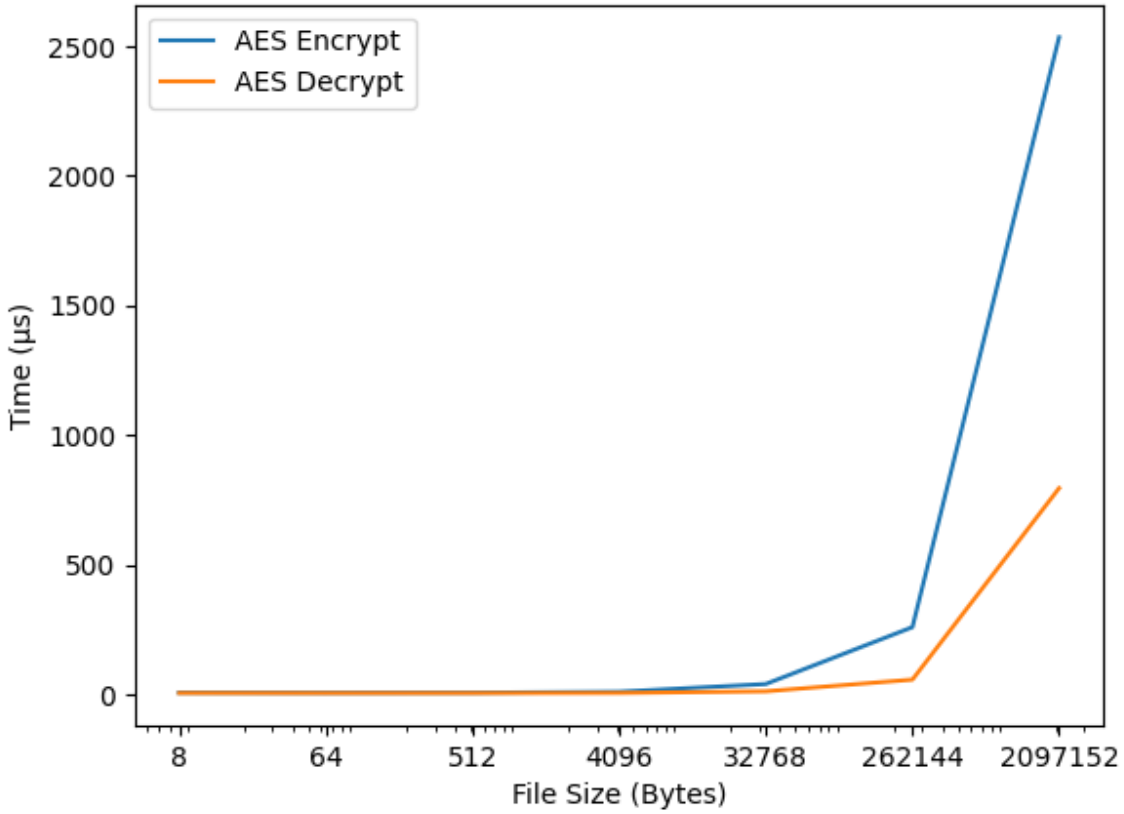

Fig. 1. AES encryption and decryption times across file sizes.

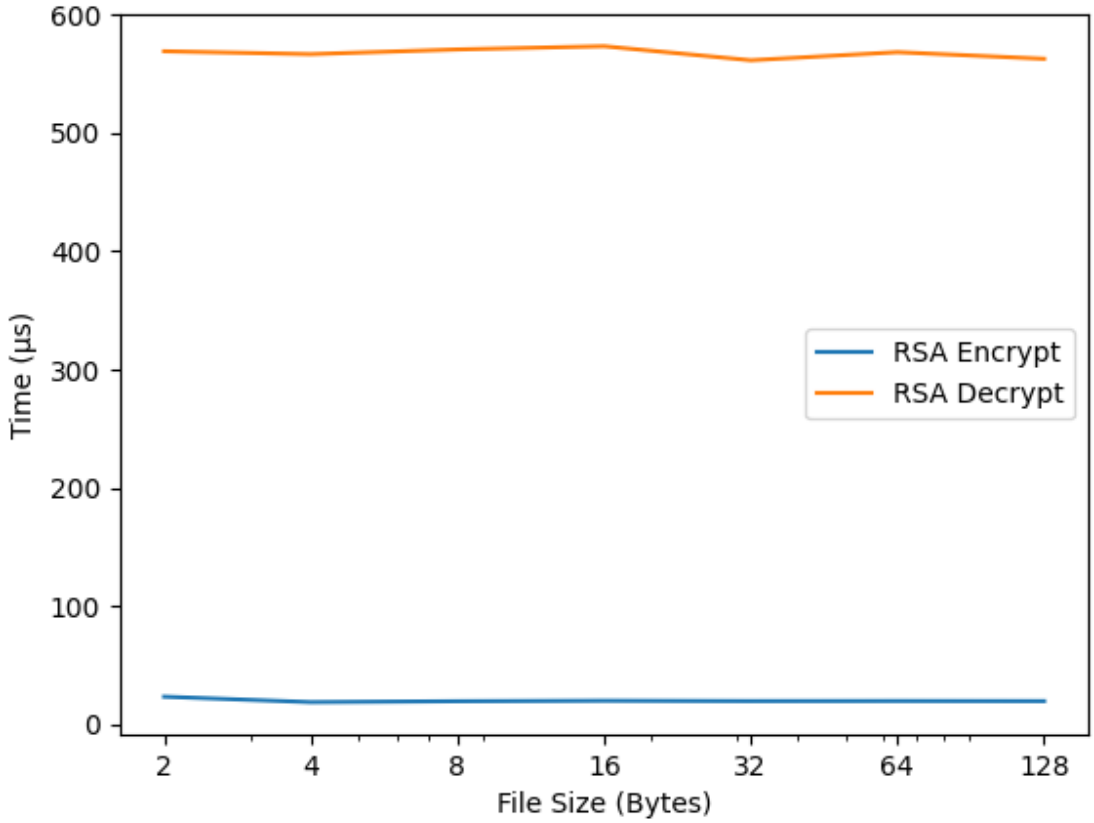

Fig. 2. RSA encryption and decryption times across file sizes.

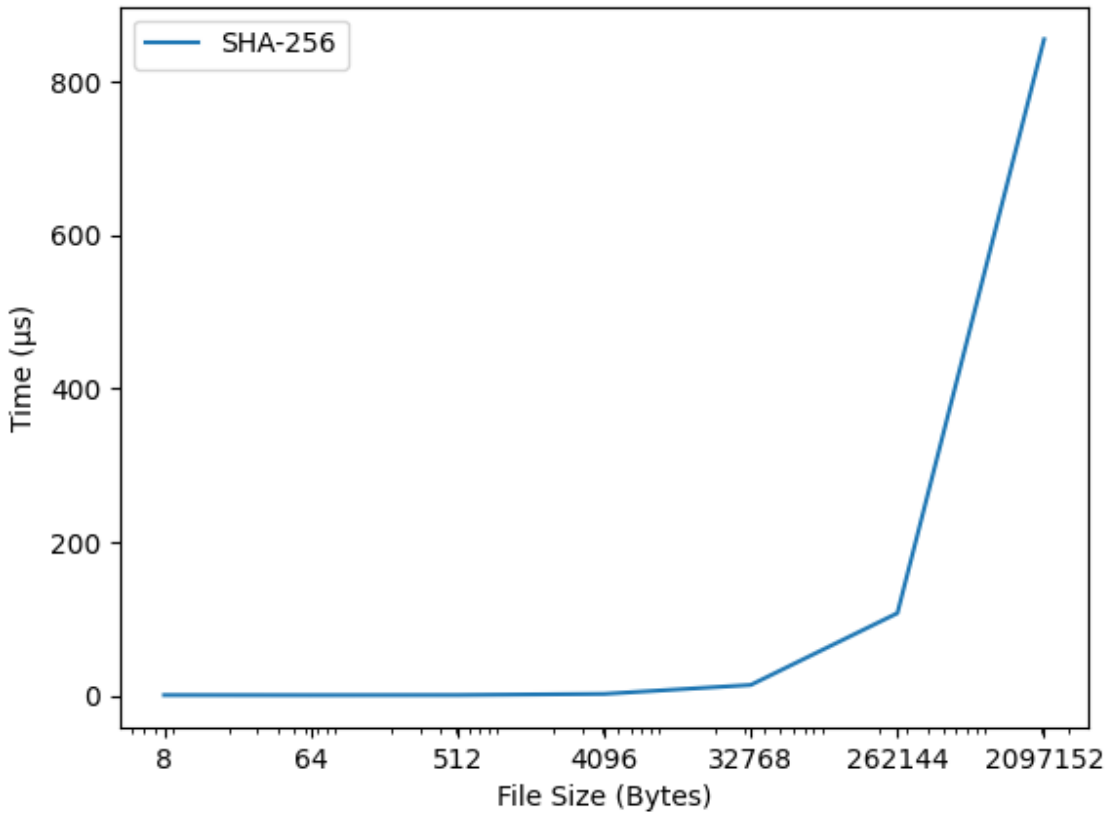

Fig. 3. SHA-256 hashing time vs. file size.