Sri Lanka Institute of Information Technology

IE3082

Cryptography

Group Assignment
Cryptographic Algorithm Evaluation (AES, RSA, SHA-256, MD5)

Year 03 Semester 01
Cyber Security
18.10.2024

Name	IT Number	Contribution Description
Piyarathna R.S.	IT22558114	Prepared the introduction, provided an overview of RSA, and documented the final report. Assisted in team coordination and quality assurance to ensure report completeness.
Jayasekera H.D.	IT22000958	Implemented AES and RSA encryption algorithms. Conducted extensive testing to validate encryption effectiveness and documented results. Contributed to code optimization and debugging.
Ranwala R.M.C.D.	IT22604408	Researched and compiled an overview of SHA-256 and MD5 encryption methods, highlighting key features and differences. Provided a detailed analysis for report inclusion.
Thilakarathna S.T.D.	IT22578914	Implemented SHA-256 and MD5 encryption algorithms. Conducted testing for encryption processes and documented findings, ensuring accuracy and consistency of results.
Siriwardhane H.H.D.V.	IT22291646	Analyzed AES encryption, providing a comprehensive overview. Summarized findings and contributed to the final report's conclusion. Reviewed the entire document for consistency and clarity.

Table of Contents

1. Introduction

This research aims to assess the performance, attributes, and practical uses of different cryptographic algorithms. Cryptography is essential for maintaining the confidentiality, integrity, and validity of data in modern digital communication. By comprehending the advantages and disadvantages of diverse cryptographic methods, we may more accurately assess their suitability for various practical applications.

We focus on hash functions, symmetric key algorithms, and asymmetric key algorithms. Each category uses a different approach to encryption, decryption, or data integrity verification and has a different security function. To have a thorough understanding of how these algorithms function in various scenarios, the evaluation will combine theoretical analysis with real-world application.

The four selected algorithms for this evaluation are:

1. AES (Advanced Encryption Standard)
2. RSA (Rivest-Shamir-Adleman)
3. SHA-256 (Secure Hash Algorithm)
4. MD5 (Message Digest Algorithm 5)

In this report, we will compare AES vs. RSA to analyze the differences between symmetric and asymmetric encryption in terms of speed, security, and computational requirements. Similarly, we will compare SHA-256 vs. MD5 to evaluate their effectiveness in ensuring data integrity, examining factors like collision resistance and hashing speed.

Through this evaluation and comparison, we aim to analyze the history, design principles, vulnerabilities, and performance characteristics of each algorithm. The results will help illustrate the trade-offs involved in choosing a cryptographic method for specific security requirements.

2. Algorithm Overview

I) AES (Advanced Encryption Standard) Algorithm

AES is a symmetric encryption algorithm followed with the aid of the National Institute of Standards and Technology (NIST) in 2001 because it was the successor to the Data Encryption Standard (DES). It turned into advanced via Belgian cryptographers Vincent Rijmen and Joan Daemen, who to begin with submitted it underneath the name "Rijndael" for the duration of NIST's opposition to find a secure and efficient alternative for DES. After a rigorous assessment technique, Rijndael changed into selected from amongst 15 competing algorithms because of its advanced security, velocity, and versatility.

Design Principles

AES is constructed on principles of substitution-permutation networks (SPN) to reap confusion and diffusion. It makes use of key sizes of **128, 192, or 256** bits and operates on **128-bit** facts blocks. The center technique entails multiple rounds of 4 important operations: SubBytes (substitution using S-field), ShiftRows (permutation), MixColumns (diffusion), and AddRoundKey (XOR with a spherical key). The quantity of rounds relies upon on the important thing **size—10 rounds for 128-bit keys, 12 for 192-bit, and 14 for 256-bit**.AES's design ensures resistance to cryptanalytic attacks whilst maintaining computational performance.

Common Use Cases

AES is extensively used in diverse industries for securing sensitive statistics. Some of its common use instances include securing communications (SSL/TLS for HTTPS), file encryption (e.g., disk encryption equipment like BitLocker and VeraCrypt), and community safety (e.g., wi-fi networks through WPA2 and WPA3 encryption). AES is likewise implemented in protocols consisting of IPsec and OpenVPN for steady VPN communication.

Vulnerabilities & Performance

Although AES itself is considered steady, vulnerabilities can arise from wrong implementations or utilization. For example, side-channel assaults like timing assaults and differential energy analysis make the most physical traits of hardware leak records about the encryption keys. Additionally, poor key control or insecure modes of operation (which include ECB) can weaken AES's effectiveness, leading to safety breaches.

AES is understood for its strong stability between safety and overall performance. It is efficient in each hardware and software, especially when optimized with hardware acceleration technology like Intel's AES-NI, which extensively boosts its speed. AES-128 is frequently desired for packages that require better performance, whilst AES-256 is desired to be used in cases requiring stronger protection.

II) RSA (Rivest-Shamir-Adleman) Algorithm

An asymmetric encryption technique called RSA was first presented by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977. Sensitive data encryption, digital signatures, and data transfer security are among its many applications. The basis for RSA's security is the mathematical challenge of factoring big composite integers. A public key for encryption and a private key for decryption are the two keys used by the algorithm.

Design Principles

RSA's security is based on the challenge of factorizing a large number n, which is the product of two prime numbers p and q. The key generation process includes selecting these primes, computing $n=p\times q$, and determining two exponents: a public exponent e and a private exponent d, which satisfy specific mathematical properties. Encryption transforms a message m into cipher text $c=m^e \bmod n$, while decryption retrieves the original message with $m=c^d \bmod n$.

Common Use Cases

RSA is widely used in:

- **Secure Data Transmission**: It plays a crucial role in HTTPS and other secure communication protocols.
- **Digital Signatures**: It verifies the authenticity and integrity of digital documents.
- **Email Encryption**: Protects email communications with encryption standards like PGP.

Vulnerabilities and Performance

A key size of less than 2048 bits makes RSA susceptible to attacks. The development of quantum computers presents a threat in the future, even though it is thought to be safe from classical computing. Because RSA requires a lot of processing, it is less effective at encrypting big data sets.

III) SHA-256 (Secure Hash Algorithm 256-bit)

The National Security Agency (NSA) unveiled SHA-256, a cryptographic hash function that belongs to the SHA-2 family, in 2001. It is a well-liked option for protecting sensitive data and guaranteeing data integrity since it can produce a fixed 256-bit hash value from an input of any length. SHA-256 is widely used in applications such as blockchain technology, digital signatures, password hashing, and secure web communications.

Design Principles

The algorithm processes input data in 512-bit blocks, using bitwise operations and modular arithmetic to transform the data iteratively. The result is a 256-bit output, and even minor changes to the input produce significantly different hash values (avalanche effect). Its design ensures collision resistance, preimage resistance, and computational infeasibility of reversing the hash.

Common Use Cases

SHA-256 is extensively utilized in:

- **Blockchain Technology**: Securing transaction data and verifying blocks.
- **Digital Signatures**: Ensuring the authenticity of documents and communications.
- **Password Hashing**: Safely storing hashed passwords for authentication.
- **SSL/TLS**: Verifying data integrity in secure web communications.

Vulnerabilities and Performance

SHA-256 is now thought to be safe from classical attacks, but quantum computing risks could make it less resilient in the future. It provides a better mix between security and performance, although it requires more computing power than more traditional algorithms like MD5.

IV) MD5 (Message Digest Algorithm 5)

Ronald Rivest first proposed the MD5 cryptographic hash function in 1991 as an improvement on the MD4 hash function. From an input of any length, usually a 32-character hexadecimal number, it generates a 128-bit hash value. MD5 was designed to generate a digital fingerprint of data quickly, making it popular for file integrity checks, password hashing, and digital signatures in its early years.

Design Principles

The algorithm processes input data in 512-bit blocks and produces a fixed 128-bit output. It consists of four rounds of 16 operations, utilizing bitwise operations and non-linear functions to transform the data. MD5's design focused on speed and simplicity, making it efficient for large datasets. However, its collision resistance is limited, allowing different inputs to produce the same hash value, which has rendered it unsuitable for cryptographic security.

Common Use Cases

MD5 was widely used for verifying file integrity, creating digital signatures, and storing hashed passwords. Today, its usage is mostly limited to non-critical applications like checksums for file downloads and basic hash-based indexing, due to its vulnerability to collision attacks.

Vulnerabilities and Challenges

The main flaw in MD5 is its vulnerability to collision attacks, in which two different inputs generate the same hash. Researchers showed this in 2004, which led to its abandonment for secure applications like digital signatures and SSL certificates. The security issues of MD5 have led to a move towards more secure alternatives like SHA-256, despite its speed and low resource use.

3. Implementation and Initial Testing

Environmental Setup

The implementation was done in Python with the pycryptodome package, which supports a wide range of cryptographic techniques. It was installed using pip.

pip install pycryptodome

Implementation Details

1) AES Implementation (128-bit Key, CBC Mode)

The AES algorithm was implemented in Cipher Block Chaining (CBC) mode, which encrypts each block based on the previous block, making it more secure than ECB mode. The following steps are included in the implementation:

- To assure randomness during encryption, a 16-byte random key and initialization vector (IV) were constructed for AES-128.
- Padding: The message was padded to match AES's block size (16 bytes).
- AES encryption and decryption were carried out using the CBC mode.
- The encryption and decryption times were measured using Python's time.perf_counter() ensures excellent accuracy.

```
PS C:\Users\Himasha\Downloads\CRYPTO> python AES.py
Encrypted (AES): b'\xe0\xf9-\xde/\xcdR-\n&\xdd\xbeC\x8e|\xde\x10V\xaa!\xd6\xb7Uc\xfd\x8e&\xb8\x99\xa1'
8b>q\x8c'...
Decrypted (AES): This is a sample message for AES encryption.
AES Encryption Time: 0.00004030 seconds
AES Decryption Time: 0.00001020 seconds
```

2) RSA Implementation (2048-bit Key)

The RSA algorithm was implemented using the RSA library. It uses asymmetric encryption, with the public key encrypting the message and the private key decrypting it. The following steps are included in the implementation:

- Key Generation: A 2048-bit RSA key pair (public and private) was created.
- Encryption/Decryption: The message was encrypted with the public key and decrypted with the private key.
- Encryption and decryption times were measured over time.Perf_counter() provides exact measuring.

```
PS C:\Users\Himasha\Downloads\CRYPTO> python RSA.py
Encrypted (RSA): b'T\xe8\xfcg\xd1\xc3#*\xd3\x93\xeb\xea\x92\xca.\xabW\x96\xfd\xa8\xd
xbet_\xf1\xa0cQ\xc2\xe0'...
Decrypted (RSA): This is a sample message for RSA encryption.
RSA Encryption Time: 0.01104750 seconds
RSA Decryption Time: 0.00811130 seconds
```

AES.py > ...

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
import time

# Generating random AES key and IV (16 bytes for AES-128)
key = get_random_bytes(16)
iv = get_random_bytes(16)

# AES in CBC mode
cipher = AES.new(key, AES.MODE_CBC, iv)

# Sample message to encrypt
message = b'This is a sample message for AES encryption.'

# Encrypting message and measure encryption time
start_time = time.perf_counter()
ciphertext = cipher.encrypt(pad(message, AES.block_size))
encryption_time = time.perf_counter() - start_time

# Decrypting message and measure decryption time
cipher_decrypt = AES.new(key, AES.MODE_CBC, iv)
start_time = time.perf_counter()
decrypted_message = unpad(cipher_decrypt.decrypt(ciphertext), AES.block_size)
decryption_time = time.perf_counter() - start_time

# Output
print(f"Encrypted (AES): {ciphertext[:50]}...")
print(f"Decrypted (AES): {decrypted_message.decode('utf-8')}")
print(f"AES Encryption Time: {encryption_time:.8f} seconds")
print(f"AES Decryption Time: {decryption_time:.8f} seconds")
```

Figure 1:AES Implementation

RSA.py > ...

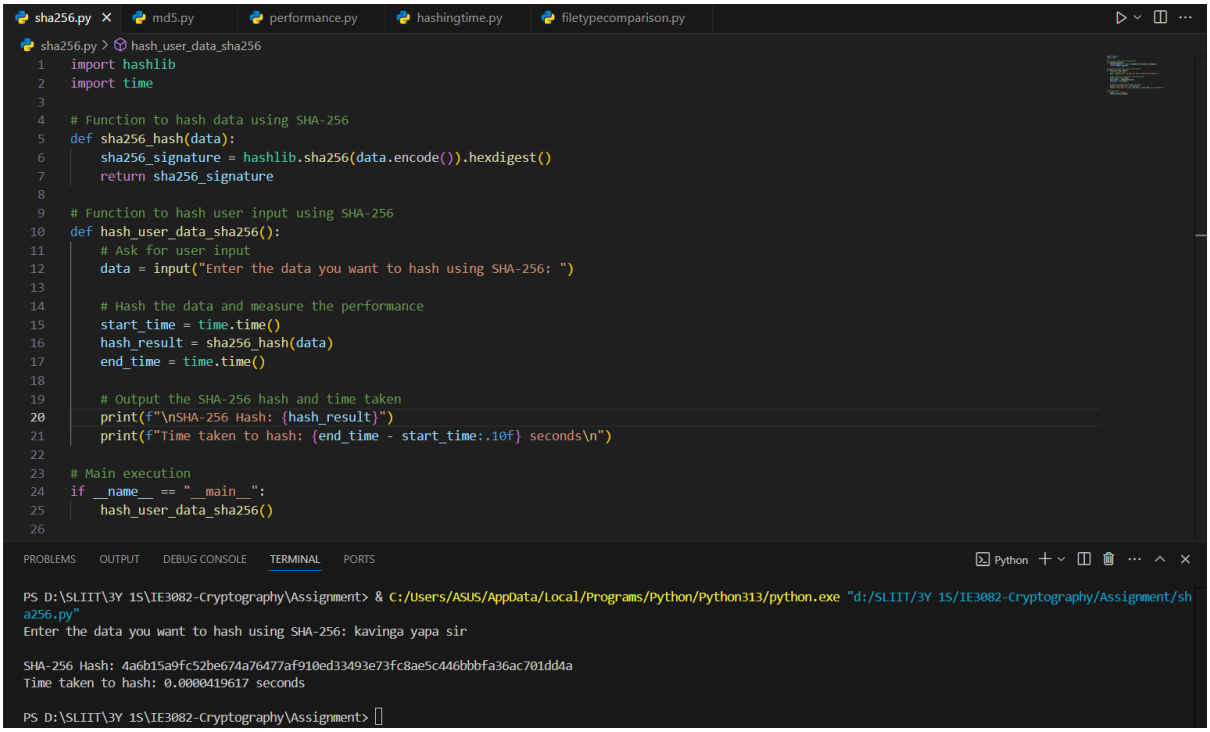
```python
import rsa
import time

# Generating RSA public and private keys (2048 bits)
(public_key, private_key) = rsa.newkeys(2048)

# Sample message to encrypt
message = b'This is a sample message for RSA encryption.'

# Encrypting message and measure encryption time
start_time = time.perf_counter()
encrypted_message = rsa.encrypt(message, public_key)
encryption_time = time.perf_counter() - start_time

# Decrypting message and measure decryption time
start_time = time.perf_counter()
decrypted_message = rsa.decrypt(encrypted_message, private_key)
decryption_time = time.perf_counter() - start_time

# Output
print(f"Encrypted (RSA): {encrypted_message[:50]}...")
print(f"Decrypted (RSA): {decrypted_message.decode('utf-8')}")
print(f"RSA Encryption Time: {encryption_time:.8f} seconds")
print(f"RSA Decryption Time: {decryption_time:.8f} seconds")
```

Figure 2:RSA Implementation

3) SHA-256 Implementation

The SHA-256 implemented using built-in “hashlib” library and “time” library used to measure the performance of the hashing Operation. It operates in the following steps:

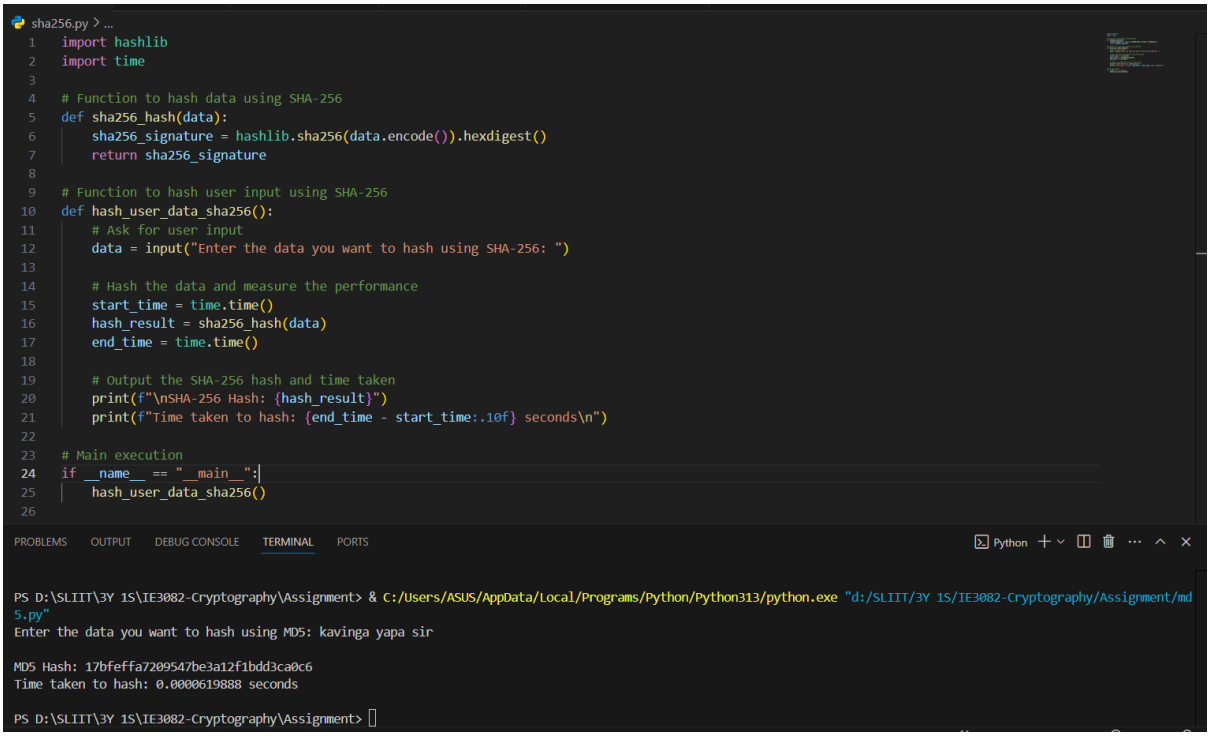
- The user is prompted to enter a string by the application.
- Once it calculates the SHA-256 hash of the input string with the sha256 hash () function
- It measures the time taken (end_time- start_time) to perform the hashing operation
- Finally, it prints the generated SHA-256 hash and the time taken to calculate it



4) MD5 Implementation

The MD5 is also implemented using the built-in “hashlib” library, and the “time” library is used to measure the performance of the hashing operation. The process works in these steps:

- The application asks the user to input a string.
- The input's MD5 hash is computed using the md5_hash () function.
- Then, it measures how long it takes to perform the hashing using time.time().
- The user sees the MD5 hash along with the time it took to compute it.



```python
import hashlib
import time

# Function to hash data using SHA-256
def sha256_hash(data):
    sha256_signature = hashlib.sha256(data.encode()).hexdigest()
    return sha256_signature

# Function to hash user input using SHA-256
def hash_user_data_sha256():
    # Ask for user input
    data = input("Enter the data you want to hash using SHA-256: ")

    # Hash the data and measure the performance
    start_time = time.time()
    hash_result = sha256_hash(data)
    end_time = time.time()

    # Output the SHA-256 hash and time taken
    print(f"\nSHA-256 Hash: {hash_result}")
    print(f"Time taken to hash: {end_time - start_time:.10f} seconds\n")

# Main execution
if __name__ == "__main__":
    hash_user_data_sha256()
```

```
PS D:\SLIIT\3Y 1S\IE3082-Cryptography\Assignment> & C:/Users/ASUS/AppData/Local/Programs/Python/Python313/python.exe "d:/SLIIT/3Y 1S/IE3082-Cryptography/Assignment/md
5.py"
Enter the data you want to hash using MD5: kavinga yapa sir

MD5 Hash: 17bfeffa7209547be3a12f1bdd3ca0c6
Time taken to hash: 0.0000619888 seconds

PS D:\SLIIT\3Y 1S\IE3082-Cryptography\Assignment>
```

4. Comprehensive Testing and Performance Analysis

Performance Comparison of AES and RSA

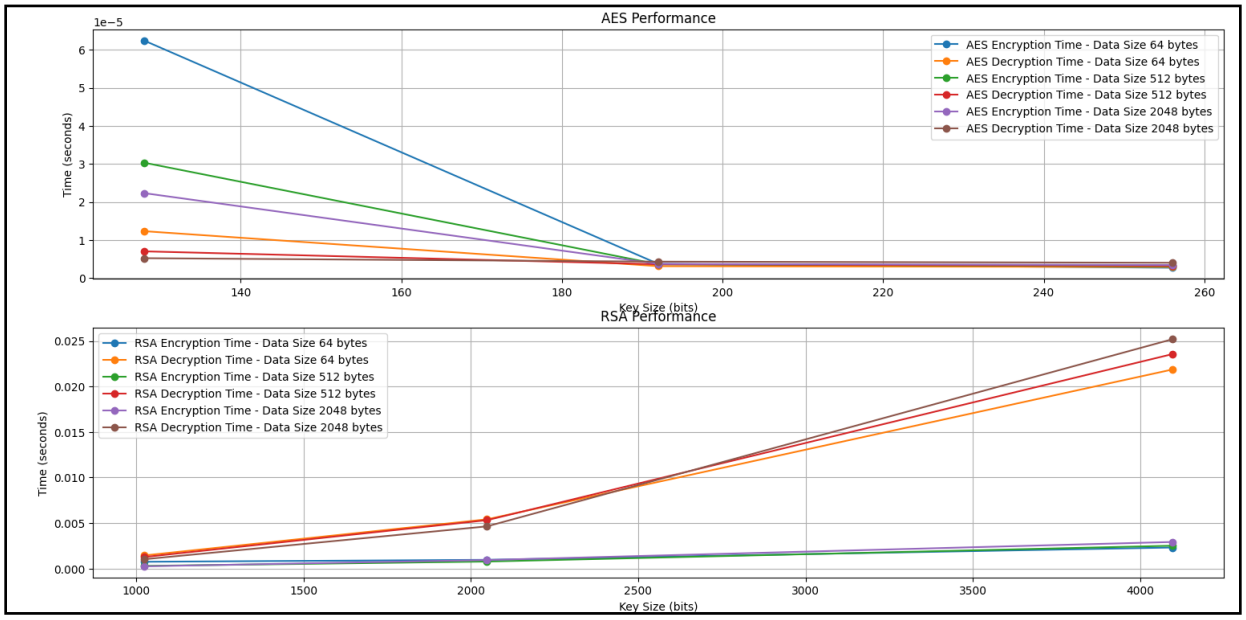
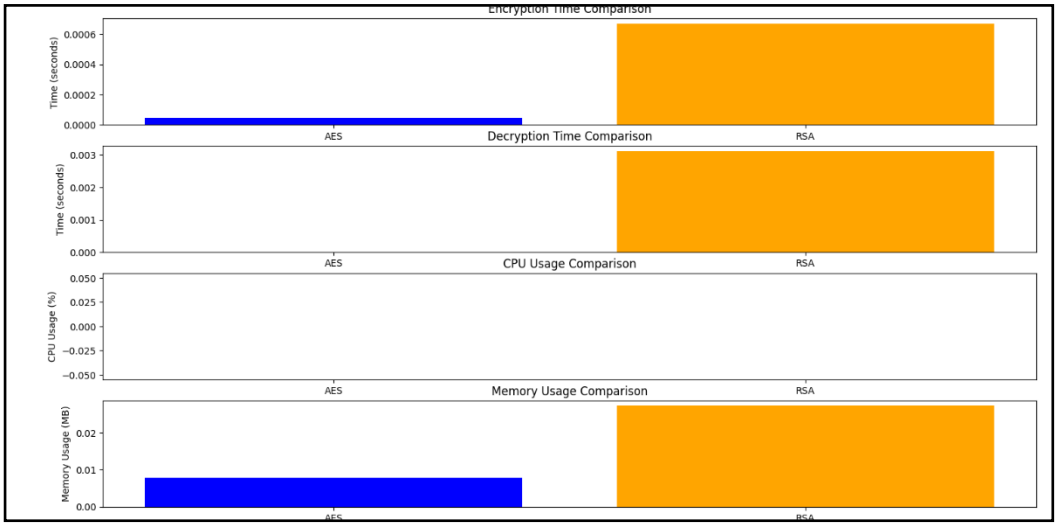


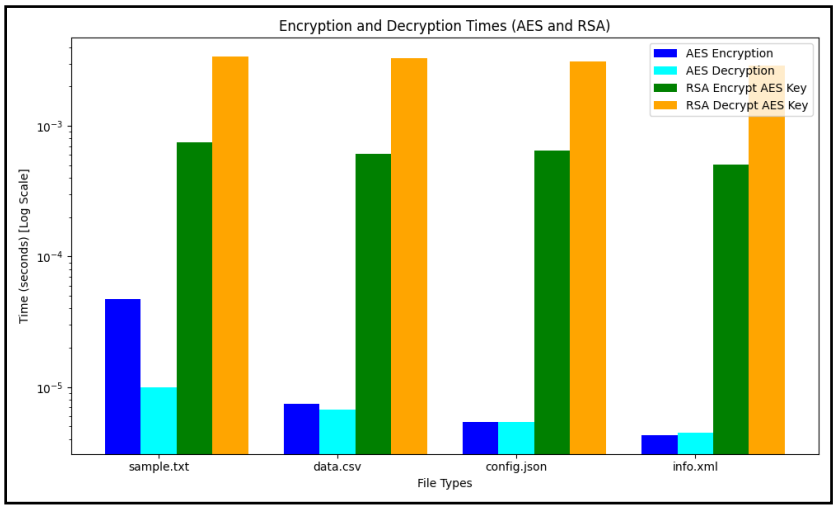
Figure 3:Key and data size impact

Encryption and Decryption time: According to the analysis, there are differences between the encryption and decryption times of AES and RSA. While RSA's encryption times somewhat increase with bigger key sizes, making it slower, AES's encryption times drop as key sizes increase. While RSA's decryption timings are much slower because of its computing requirements, AES's decryption durations similarly drop or stable with key size. All things considered; AES continuously performs better than RSA in terms of efficiency.

Key Size and Data size impact: The above graph depicts that key and data size influence AES and RSA performance. AES performs better with bigger key sizes, particularly for smaller data volumes, despite the additional computational complexity. RSA, on the other hand, sees significant increases in encryption and decryption times with bigger key sizes, indicating greater operational complexity. AES easily handles greater data sizes, but RSA is sensitive to data size, causing it unsuitable for direct large data encryption.



The above comparison of AES and RSA reveals that AES greatly exceeds RSA in terms of encryption and decryption times, with AES taking almost no time for both processes, while RSA is significantly slower. AES also requires less memory than RSA, making it more efficient in terms of resource utilization. Both algorithms use minimal CPU resources, however AES is significantly better optimized. Overall, AES is better suited for situations that require rapid, efficient encryption and low resource consumption, whereas RSA, despite being slower and more resource-intensive, is preferable for secure key exchanges due to the usage of asymmetric keys.

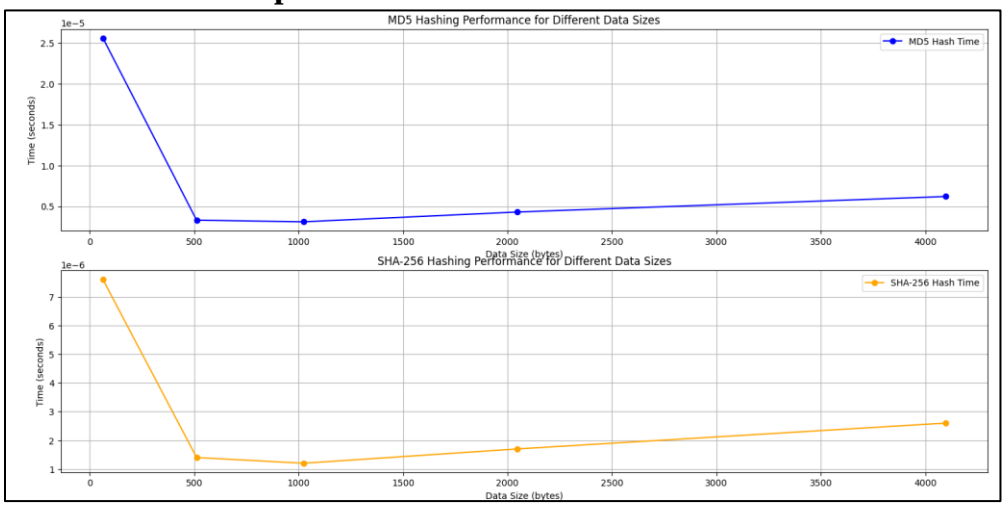



```
Processing file: sample.txt
AES Encryption Time: 0.0000471000s, AES Decryption Time: 0.0000099000s
RSA Encryption Time: 0.0007497000s, RSA Decryption Time: 0.0033830000s

Processing file: data.csv
AES Encryption Time: 0.0000074001s, AES Decryption Time: 0.0000067001s
RSA Encryption Time: 0.0006070001s, RSA Decryption Time: 0.0032814000s

Processing file: config.json
AES Encryption Time: 0.0000054000s, AES Decryption Time: 0.0000053999s
RSA Encryption Time: 0.0006437000s, RSA Decryption Time: 0.0031059000s

Processing file: info.xml
AES Encryption Time: 0.0000042999s, AES Decryption Time: 0.0000044999s
RSA Encryption Time: 0.0005038000s, RSA Decryption Time: 0.0028730000s
```

The bar chart above compares the encryption and decryption timings for AES and RSA on four distinct file types with the same data size: sample.txt, data.csv, config.json, and info.xml. The speeds are displayed on a logarithmic scale, indicating that AES encryption and decryption (blue and cyan) are substantially faster than RSA encryption and decryption (green and orange). While

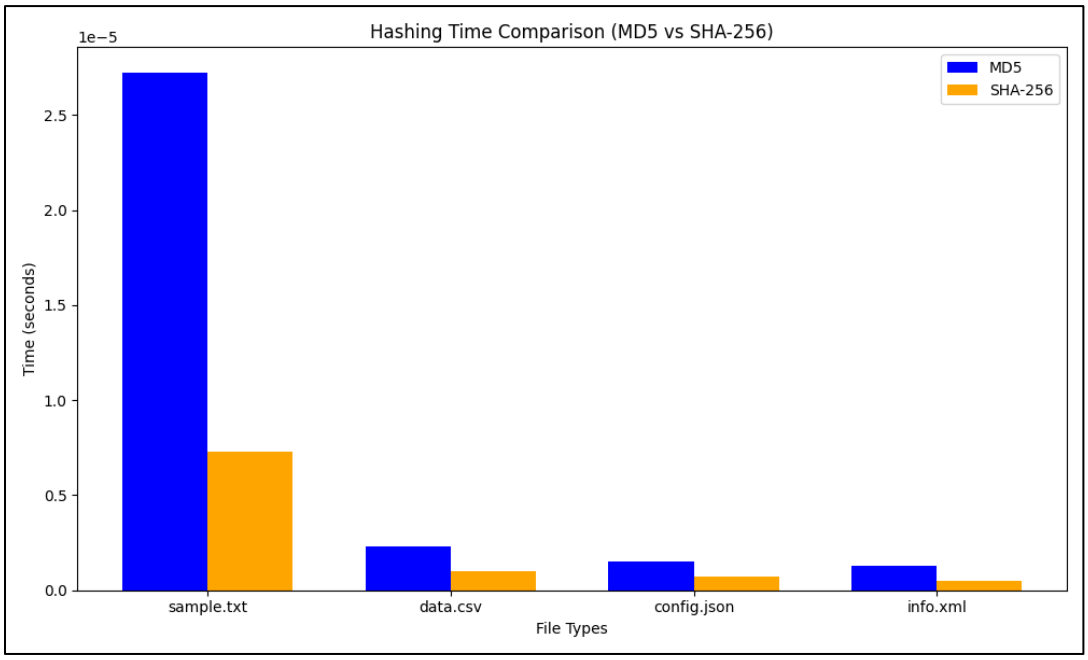
AES encryption and decryption speeds are typically in the 10^−5 to 10^−4 second range, RSA encryption of the AES key takes longer, about 10^−4 to 10^−3 seconds, and RSA decryption of the AES key is the slowest, lasting more than 10^−3 seconds for all file types. The results highlight AES's efficiency for data encryption and RSA's rather poor performance, especially in key management tasks.

Performance Comparison of SHA-256 and MD5



The two graphs above show how the MD5 and SHA-256 hashing algorithms perform (time) with various data sizes. MD5 usually runs quicker than SHA-256, showing more consistent and stable outcomes for different data sizes. SHA-256 has more noticeable performance fluctuations, especially with smaller data sizes, but it is still slower because of its more secure and complex hashing process. Both algorithms demonstrate a decrease in hashing time when the data size initially increases, but MD5 proves to be more efficient overall for larger data sets.

The three bar charts above compare the MD5 and SHA-256 algorithms in terms of hashing time, CPU utilization, and memory usage for a 4096-byte data set. According to first bar chart, SHA-256 is quicker than MD5 for this data size (4096 bytes), even though it's more complex. In the second bar chart, both algorithms display very low CPU usage (around 0% or less than 0.05%), indicating that the CPU impact for both MD5 and SHA-256 is minimal for this data size. The last bar chart shows that SHA-256 uses more memory than MD5, probably because its algorithm is more complex and it has a larger hash size.



The bar graph above shows a comparison of the time it takes to hash various types of files using the MD5 and SHA-256 algorithms. The chart shows that MD5 (blue) takes a lot more time than SHA-256 (orange) when processing .txt files.MD5 and SHA-256 both take less time than for .txt files, but MD5 is still a bit longer than SHA-256 when it comes to .csv files. The hashing times for both algorithms are nearly the same and quite low for .json and .xml files.

5. Conclusion

The comparative analysis of the cryptographic algorithms AES, RSA, SHA-256, and MD5 reveals key distinctions in their design, performance, and use cases. Applications like file and network security that demand quick encryption and minimal computing overhead are best suited for AES's symmetric key technique, which excels in efficiency and speed. The computationally demanding nature of the RSA method restricts its direct application for huge data encryption, but it provides strong security for applications like digital signatures and safe key exchanges.

Strong security for guaranteeing data integrity is demonstrated by SHA-256, which is frequently used in digital signatures and blockchain technology. It is more computationally demanding than older algorithms, such as MD5, which, in spite of its speed, is no longer appropriate for secure applications because of its susceptibility to collision attacks.

In conclusion, even though each algorithm has advantages, the decision is based on the particular needs for processing power, speed, and security. While RSA is mostly used to secure key exchanges in conjunction with other encryption techniques, AES and SHA-256 are still the recommended choices for encryption and hashing because of their balanced performance and security.

6. References

1. Simplilearn, "SHA 256 Algorithm Tutorial," [Online]. Available: https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm#:~:text=SHA%20256%20is%20a%20part,strength%20against%20brute%20force%20attacks. [Accessed: Oct. 10, 2024].

2. Supra Academy, "The NSA and Bitcoin: Origins of the SHA-256 Hashing Algorithm," [Online]. Available: https://supra.com/academy/the-nsa-and-bitcoin-origins-of-the-sha-256-hashing-algorithm/. [Accessed: Oct. 10, 2024].

3. J. Singh, A. Sharma, and S. Kaur, "Analysis of Secured Hash Algorithm-256 in a Blockchain-Based Money Transaction System," 2023 International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems (ICSES), Chennai, India, 2023, pp. 1-6, doi: 10.1109/ICSES60034.2023.10465578.

4. R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, Apr. 1992.

5. B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. New York, NY, USA: Wiley, 1996.

6. X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," in *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, 2005, pp. 19-35.

7. M. Cobb, "RSA Algorithm: Secure Your Data with Public-Key Encryption," Simplilearn, [Online]. Available: https://www.javatpoint.com/rsa-encryption-algorithm. [Accessed: Oct. 09, 2024].

8. Okta, "RSA Encryption," [Online]. Available: https://www.okta.com/identity-101/rsa-encryption/. [Accessed: Oct. 09, 2024].

9. National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)," FIPS PUB 197, 2001.

10. J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Berlin, Germany: Springer, 2002.

11. B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 3rd ed. New York, NY, USA: Wiley, 2015.

12. N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering: Design Principles and Practical Applications*. Hoboken, NJ, USA: Wiley, 2010.

13. IEEE 802.11, "Wi-Fi Security: WPA2 and WPA3," IEEE Press, 2018.

14. P. C. Kocher, et al., "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, 1996, pp. 104-113.

15. P. Kocher, et al., "Differential Power Analysis," in *Proceedings of CRYPTO'99*, 1999.

16. S. Gueron, "AES-NI Performance Evaluation on Recent Intel Processors," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 85-93, June 2012.