Sri Lanka Institute of Information Technology

Information Security Risk Management Assignment

Selected Algorithms: AES, RSA, SHA-256

Group_03		
Name	**IT Number**	**Contribution Description**
J. Aazaf Ritha	IT23151710	Managed overall team coordination and structured the complete report. Led the AES encryption/decryption implementation, testing, and optimization. Conducted performance testing, analysis, and documentation of AES design, vulnerabilities, and risk mitigation. Assisted with tests for AES, RSA, and SHA-256. Reviewed and integrated teammates' code for RSA and SHA-256 to ensure consistency.
S.M.F. Hana	IT23255142	Drafted RSA section including history, design principles, vulnerabilities, and performance evaluation. Assisted in analyzing results and ensuring testing accuracy. Supervised RSA implementation and performance testing.
M.F.M. Farhan	IT23422070	Prepared the SHA-256 section covering history, design, vulnerabilities, and performance. Developed and implemented SHA-256 hashing in Python. Collaborated on graphs, performance analysis, and code refinement to ensure accurate results and secure implementation.
H.M.S.H Wijerathna	IT23306936	Implemented and tested RSA encryption/decryption in Python. Conducted comprehensive testing and performance analysis. Created graphs and visualizations for encryption/decryption time comparisons and contributed results to the final report.
R.M.T.P. Rasnayake	IT23246546	Developed AES encryption/decryption code in Python. Assisted in debugging, code refinement, and testing. Supported performance measurement and contributed to AES documentation and final report preparation.

Module: IE 3082 – Cryptography

Year 3, Semester 1 (2025) - B.Sc. (Hons) in Information Technology Specialized in Cyber Security

1 Executive Summary

This report evaluates three fundamental cryptographic algorithms: AES (Advanced Encryption Standard), RSA (Rivest–Shamir–Adleman), and SHA-256. The study aims to assess their design principles, performance, security strengths, and suitability for real-world applications.

AES was chosen as the symmetric key algorithm for its efficiency and strong adoption in securing bulk data. RSA was selected as the asymmetric algorithm due to its role in secure key exchange and digital signatures. SHA-256 was included as the hash function for its reliability in ensuring data integrity and its extensive use in applications such as blockchain.

The work involved three main stages: (1) researching the theoretical background of each algorithm, (2) implementing and testing their functionality, and (3) conducting performance analysis under different conditions, including variations in key size, data size, and computational load.

Findings show that AES is the fastest and most efficient for large-scale encryption, RSA provides high security for authentication and key management but at a higher computational cost, and SHA-256 delivers consistent integrity verification with low resource usage. The combined use of these algorithms reflects best practices in modern cryptographic systems, offering confidentiality, authenticity, and integrity.

2 Research on Selected Algorithms

2.1 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) was standardized by the U.S. National Institute of Standards and Technology (NIST) in 2001 as the successor to the aging Data Encryption Standard (DES). AES was designed by Vincent Rijmen and Joan Daemen under the name *Rijndael* and selected through an open international competition. Unlike DES, which used a 56-bit key, AES supports key lengths of 128, 192, and 256 bits, offering a significantly higher security margin. AES operates on fixed 128-bit data blocks, performing 10, 12, or 14 transformation rounds depending on the key size [1]. Since its adoption, AES has become the most widely deployed symmetric encryption standard worldwide.

Design Principles

AES is based on a Substitution–Permutation Network (SPN) structure, which ensures both confusion and diffusion in the ciphertext. Each round of AES (except the first and last) applies four transformations:

- SubBytes: a non-linear substitution step using S-Boxes.
- ShiftRows: a cyclic row shift that ensures diffusion.
- MixColumns: a linear mixing operation that combines bytes in each column using matrix multiplication over a Galois Field.
- AddRoundKey: XOR of the state with a round key generated by the key schedule.

The number of rounds is determined by key size: 10 rounds for AES-128, 12 for AES-192, and 14 for AES-256. Unlike DES, AES processes the entire 128-bit state in each round, contributing to its efficiency in both software and hardware implementations [2].

Applications

AES is the backbone of modern confidentiality systems and is integrated into numerous international standards. It is used in TLS/SSL for web security, IPsec for secure internet communications, WPA2/WPA3 for Wi-Fi encryption, SSH for secure remote access, and disk encryption solutions such as Microsoft BitLocker and Apple FileVault. AES is also deployed in secure voice and video applications like Skype, as well as countless commercial security products [3].

Known Vulnerabilities

While AES is cryptographically secure, improper implementation can introduce weaknesses. Side-channel attacks (timing or power analysis) may leak key information if hardware or software is not protected. Using insecure modes like ECB can also reveal plaintext patterns. Strong key management, secure padding, and authenticated modes (e.g., GCM**,** CBC-MA) are essential to maintain AES security [4] .

Risks and Countermeasures

1. Side-Channel Attacks: Exploit timing or power variations to recover keys. *Countermeasure*: Use side-channel-resistant hardware and constant-time cryptographic operations.
2. Weak Operation Modes (e.g., ECB): Exposes data patterns in ciphertext. *Countermeasure*: Use secure modes such as CBC, CTR, or GCM.
3. Poor Key Management: Reused or weak keys can compromise encryption. *Countermeasure*: Use CSPRNGs, periodic key rotation, and store keys securely (e.g., in HSMs).
4. Quantum Threats: Quantum computing could reduce AES's effective key strength. *Countermeasure*: Prefer AES-256 for long-term post-quantum resilience.

Performance Characteristics

AES is highly optimized for performance. Its substitution–permutation structure enables parallel processing and hardware acceleration. AES-128 is the fastest, followed by AES-192 and AES-256, which trade a small speed reduction for stronger security. On modern systems, AES can process data at several hundred megabytes per second, making it suitable for real-time and large-scale applications.

2.2 Rivest–Shamir–Adleman (RSA)

The RSA (Rivest–Shamir–Adleman) cryptosystem, introduced in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman, represents a foundational breakthrough in modern cryptography. Emerging shortly after Whitfield Diffie and Martin Hellman's pioneering 1976 paper on public-key cryptography, RSA was the first practical implementation of an asymmetric encryption algorithm, providing a secure method for both encryption and digital signatures. Its security is rooted in the mathematical difficulty of factoring large composite numbers, specifically the product of two large prime numbers. RSA revolutionized secure digital communication by enabling key exchange and authentication without requiring prior shared secrets, laying the groundwork for modern secure protocols such as SSL/TLS, digital certificates, and electronic signatures. Despite the rise of elliptic curve cryptography and other advanced methods, RSA remains one of the most widely deployed public-key systems in both academic and commercial applications [2].

Design Principles

1. Asymmetric (Public-Key) Cryptographic Principle
   - RSA is built on the public/private key concept, allowing secure communication without pre-shared keys.
   - One key is made public for encryption and the other (private) is kept secret for decryption.

2. Mathematical Foundation: Integer Factorization Problem
   - The security of RSA relies on the difficulty of factoring the product of two large primes n=pq.
3. Key Generation Design
   - Choose two large primes p,q compute $n=pq$.
   - Compute Euler's totient $\phi(n)=(p-1)(q-1)$
   - Select e (public exponent) such that $gcd\ (e,\phi(n))=1$.
   - Compute d (private exponent) as the modular multiplicative inverse of $e\ mod\ \phi(n)$.
   - This mathematical relationship ensures $(M^e)^d \equiv M(mod\ n)$
4. Encryption & Decryption Symmetry
   - RSA uses modular exponentiation for both encryption and decryption
   - Encryption: $C=M^e\ mod\ n$
   - Decryption: $M= C^d\ mod\ n$
   - The system's design ensures reversibility because of Euler's theorem.
5. Efficiency and Practical Considerations
   - Using small public exponents like $e=65537$ speeds up encryption without weakening security.
   - Chinese Remainder Theorem reduces decryption time (CRT optimization)
   - Key length: Typically, 2048+ bits to resist modern attacks.

Application

The RSA algorithm is a foundation of modern information security, providing confidentiality, authentication, and data integrity. It is widely used in SSL/TLS for secure online communication, digital signatures for verifying authenticity and non-repudiation, and digital certificates within PKI to authenticate websites and software vendors. In practice, RSA mainly secures key exchanges by encrypting symmetric session keys used in faster algorithms like AES. It also supports email encryption (PGP), VPNs, and software code signing. Despite earlier computational limits, advances in hardware and optimized algorithms have made RSA efficient for both software and hardware use.

Known Vulnerabilities

1. Protocol Vulnerabilities: These are implementation-level weaknesses that occur when RSA is used incorrectly in cryptographic protocols.
   - Example: Malleability attacks, where an attacker can modify a ciphertext to produce a predictable change in the plaintext.
   - Cause: Using RSA without proper padding (e.g., OAEP).
   - Mitigation: Follow modern standards like PKCS#1, which specify secure padding schemes and key handling rules.
2. Mathematical Vulnerabilities: These arise from attempts to factor the RSA modulus (n = p × q).
   - If the modulus is too small (e.g., 1024 bits or less), it becomes vulnerable to integer factorization attacks using advanced algorithms and computational power.
   - Impact: Once n is factored, the attacker can compute the private key.
   - Mitigation: Use sufficiently large key sizes (2048–4096 bits) to ensure factoring remains computationally infeasible.
3. Side-Channel Vulnerabilities: These are physical attacks that exploit leaked information from hardware during RSA operations.
   - Examples: Timing attacks, power analysis (SPA/DPA), and fault injection.

- In the excerpt, the simple power analysis (SPA) example shows how observing variations in power consumption can reveal bits of the private key.
- Mitigation: Implement countermeasures such as constant-time algorithms, dummy operations, and noise generation.

Performance Characteristics

RSA's performance is influenced by key size, computational complexity, and operation type. Encryption and verification are relatively fast with smaller exponents, while decryption and signature generation are slower due to large private exponents. Its performance decreases with larger key sizes, as security increases at the cost of speed. To balance efficiency, RSA is often used only for key exchange or digital signatures, while symmetric algorithms handle bulk data encryption.

2.3 Secure Hash Algorithm 256 (SHA-256)

The NSA created SHA-256, which is a member of the SHA-2 family, which was standardized by NIST in FIPS 180-2 (2001) and updated in FIPS 180-4 (2015). After collision concerns, it replaced SHA-1 and is now widely used in real-world systems, such as TLS stacks, software package managers, and code-signing pipelines. [2] [5] It also serves as the foundation for many blockchain designs, such as Bitcoin, which double-hashes block headers and transactions using SHA-256.

Design Principles

The internal structure of SHA-256 is a Merkle-Damgård construction over 512-bit message blocks with padding and length encoding. [2] Each block is processed by a 64-round compression algorithm that is based on 32-bit additions, rotations, and XORs, in addition to the Choice and Majority boolean functions. Fractional square roots of the first primes are used to create the initial state (IV), and fractional cube roots are used to generate the 64 round constants. The ultimate output is a fixed 256-bit digest that provides about 2^{128} collision security. [2] [5]

Application

Common uses include "hash-then-sign" processes, in which arbitrary messages are first hashed, and the digest is signed with RSA/ECDSA/EdDSA to produce compact, fixed-size signatures, and integrity and authentication using HMAC-SHA-256. [2] Additionally, it is used for wide integrity checks in distributed systems and blockchains, Merkle trees and proofs, and data fingerprinting.

Known Vulnerabilities

There are no known vulnerabilities from a security perspective preimage resistance is approximately 2^{256} operations, and collision resistance is approximately 2^{128}. The primary warnings pertain to its usage Raw SHA-256 should not be used as a MAC or used alone for password storage since it is susceptible to length-extension attacks due to Merkle-Damgard. [2]

Performance characteristics

The algorithm is highly efficient, using less than one kilobyte of memory and processing large files without fully loading them. On typical processors, it can hash a few hundred megabytes per second, while modern hardware with dedicated instructions reaches speeds of one to several gigabytes per second. Performance increases with data size, as larger files take longer but are usually faster to process than many small ones because there is less setup overhead. Batching small messages or reusing hashing objects improves efficiency, and the algorithm scales well across multiple cores for near-linear speed gains. GPU acceleration helps mainly with large batches of independent data but offers little benefit for

single files. Overall, it is faster than SHA-3 on most processors, slower than BLAKE2 and BLAKE3, and slower than the insecure SHA-1 [2] [5].

3 Implementation & Initial Testing

The AES and RSA algorithms were implemented in Python 3.10 using the PyCryptodome library, which provides secure cryptographic primitives. For SHA-256, HMAC-SHA256 implementations were developed using Python's standard libraries (hashlib, hmac) along with PyCryptodome, combined into a single script for hashing and message authentication testing.

For AES two versions were developed: a text encryption program to test encryption/decryption correctness and measure timing across key sizes, and a file encryption program (tested on images) to confirm correct operation on binary data. Both used AES-256 in CBC mode with HMAC-SHA256 for data confidentiality and integrity.

```
S>python "AES Text Encryption.py"
Algorithm   Input Size   Trials   Avg Enc (s)   Avg Dec (s)   Status
------------------------------------------------------------------------
AES-128     100         B 5      0.000366      0.000057      Passed
AES-128     1.0         KB 5      0.000051      0.000055      Passed
AES-128     10.0        KB 5      0.000108      0.000102      Passed
AES-128     100.0       KB 5      0.000594      0.000585      Passed
AES-192     100         B 5      0.000042      0.000043      Passed
AES-192     1.0         KB 5      0.000042      0.000056      Passed
AES-192     10.0        KB 5      0.000096      0.000101      Passed
AES-192     100.0       KB 5      0.000724      0.000688      Passed
AES-256     100         B 5      0.000064      0.000053      Passed
AES-256     1.0         KB 5      0.000048      0.000050      Passed
AES-256     10.0        KB 5      0.000094      0.000099      Passed
AES-256     100.0       KB 5      0.000621      0.000649      Passed
```

Figure 3-1 – AES Text Encryption Performance Results

Figure 3.1 shows that encryption and decryption passed for all input sizes (100 B – 100 KB) and key lengths (128, 192, 256 bits).

```
--- User Input Encryption Demo ---
Enter a message to encrypt: This is Secret

[Encryption Result]
Key (hex): 0a8556120234bab981f4dbec3c1c450dad173d6f370340c27504440644f92ba2
IV  (hex): acc17f314156d705c2ebfa27b5c63a5f
Ciphertext (hex): eb1ae1549774b2bbdf1e83ba105f3416
HMAC Tag (hex): 8edcd3ad453d1add8e59d243938e143a0ba15f73187d3448edaab04b86f8634e
Decrypted: This is Secret
```

The program also allowed users to input plain text for AES-256 encryption. Figure 3.2 shows the user demo where the text *"Hana is good"* was encrypted and correctly decrypted.

Figure 3-2 – AES-256 Text Encryption and Decryption

The output displayed the generated random Key, IV, Ciphertext, and HMAC Tag, verifying correct and secure operation.

```
C:\Users\aazaf\OneDrive - Sri Lanka Institute of Information Technology\Y3S1
-Aazaf_ZenBook\2025 July\IE 3082 - Cryptography\Assignment\Group_03_Kandy\AE
S>python "AES Image Encryption.py"
Enter the full path of the file to encrypt: C:\Users\aazaf\Desktop\hana.jpg
AES Key (hex): e65882f95073c432970d5f52f559ed7a0ae4877a8e6115779f575ae55ad7b
15e
MAC Key (hex): bd087d89b372456879570641babe11b9d63fcbcfa1acacc80170ba67fb9a8
62e
File encrypted successfully: C:\Users\aazaf\Desktop\hana.jpg.enc
File decrypted successfully: C:\Users\aazaf\Desktop\hana.jpg_dec
```

Figure 3-3 – AES File Encryption and Decryption Output

To validate functionality on files, an image was encrypted using AES Image Encryption.py. The program generated a random 256-bit AES key and MAC key, producing two files: Encrypted file → hana.jpg.enc; Decrypted output → hana.jpg_dec

All tests proved the implementation worked correctly. HMAC verification ensured tamper protection and identical output after decryption.

For RSA, a program was developed to test key generation, encryption, and decryption using the PKCS1_OAEP padding scheme.

```
[Initial Test] RSA-2048 OAEP-SHA256 round-trip: OK
Key Size | Enc Time (ms) | Dec Time (ms) | Status
------------------------------------------------------------
RSA-1024    1024     5           0.320          1.223     Passed
RSA-2048    2048     5           0.730          4.144     Passed
RSA-4096    4096     5           2.666         22.659     Passed
```

Figure 3-4 – RSA Encryption and Decryption Performance Results

Figure 3.4 shows encryption and decryption results for key sizes 1024, 2048, and 4096 bits. Encryption time was consistently lower than decryption time, and overall time increased with key size as expected for RSA operations.

```
--- User Input Encryption Demo ---
Enter a message to encrypt: This is another secret

[Encryption Result]
Public Key (n bits): 2048
Ciphertext (hex): 10cbc368d9a180d24ef9fea70c98921c5103cb12c8022354afecb7b79ebe45b880
144d44cee2791b...
Decrypted: This is another secret
```

Figure 3-5 – RSA User Input Encryption and Decryption

The program also allowed user input testing for custom text encryption. As shown in Figure 3.5, the message successfully encrypted and decrypted using a 2048-bit RSA key, verifying correct functionality.

All tests passed successfully, validating that the RSA implementation worked correctly and securely for both fixed and user-supplied input data.

For SHA-256, a program was developed to verify both hashing and message-authentication functionality using SHA-256 and HMAC-SHA-256. The implementation was written in Python 3.10 using the standard libraries hashlib and hmac. SHA-256 ensures data integrity, while HMAC-SHA-256 provides authenticity through a shared secret key.

```
=== SHA-256 / HMAC-SHA256 Implementation & Initial Testing ===

Plaintext: Hello IE3082 - Cryptography Test
SHA-256 Digest: f3e954f8dbe372e46b9a48a53bf57f894b4cb1f7bde3a7e43a353fafa5eace30
HMAC-SHA256 Tag: cec1c6d08c438c02580df9e9a9ff3aa71a2a0da0ab53ef5a18e4c0df145d7cc4

Tamper Detection:
Tampering detected successfully.
```

Figure 3-6- SHA-256 / HMAC-SHA-256 Implementation and Tamper Detection Output

Figure 3.6 shows the output where a fixed plaintext message was hashed and an HMAC tag was generated. When a single character in the message was changed, the program produced a completely different tag, proving the ability of SHA-256 to detect any data alteration (avalanche effect). The results confirm that the algorithm accurately identifies tampering and maintains data integrity.

```
=== SHA-256 / HMAC-SHA256 User Input Demo ===
Enter a message to hash: testing hashing
SHA-256 Digest: fc9715d473fdd3a400cf492c69164e6d738c962c463885b840b0b28f333260d6
HMAC-SHA256 Tag: fdf64793bb8a39f346bc3579f9c522fb7948592a71b2f1a27ef10a5b2f89b5ce

Enter a message to verify (to test tampering): testing hashing
Message verified successfully (integrity intact).
```

Figure 3-7 – SHA-256 / HMAC-SHA-256 User Input Verification Demo

The user input demo shown in Figure 3.7 demonstrates that when the same text is entered again, the digest and HMAC tag match, verifying that message integrity is preserved.

If any modification occurs, verification fails, confirming detection of unauthorized changes.

```
=== SHA-256 / HMAC-SHA256 Performance Test ===
Input Size (bytes) | SHA-256 Avg (ms) | HMAC-SHA256 Avg (ms)
------------------------------------------------------------
100                0.0035             0.0060
1024               0.0043             0.0089
10000              0.0130             0.0184
100000             0.0982             0.1001
1000000            0.7715             0.7866
```

Figure 3-8 – SHA-256 / HMAC-SHA256 Performance Results

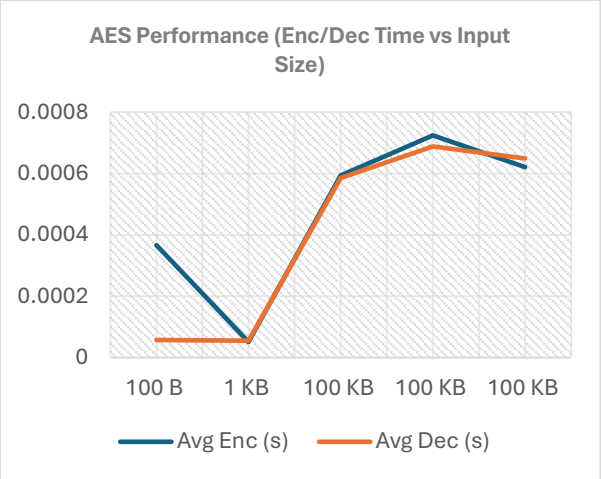
To evaluate performance, SHA-256 and HMAC-SHA-256 were tested for input sizes from **100 bytes (B)** to **10 megabytes (MB)**, using five trials each. The average processing times are summarized below.

The test results show that execution time increased linearly with input size, and HMAC-SHA-256 was slightly slower than SHA-256 because of the additional key-based hashing operation. Both completed even 10 MB inputs within 8 ms, confirming high efficiency for real-time integrity and authenticity verification.

4 Comprehensive Testing and Performance Analysis

Performance was measured for AES-128, AES-192, and AES-256 across input sizes from 100 B to 100 KB (5 trials each).

Key Size (bit)	Input Size	Avg Enc (s)	Avg Dec (s)	Status
128	100 B	0.000366	0.000057	Passed
128	1 KB	0.000051	0.000055	Passed
128	100 KB	0.000594	0.000585	Passed
192	100 KB	0.000724	0.000688	Passed
256	100 KB	0.000621	0.000649	Passed

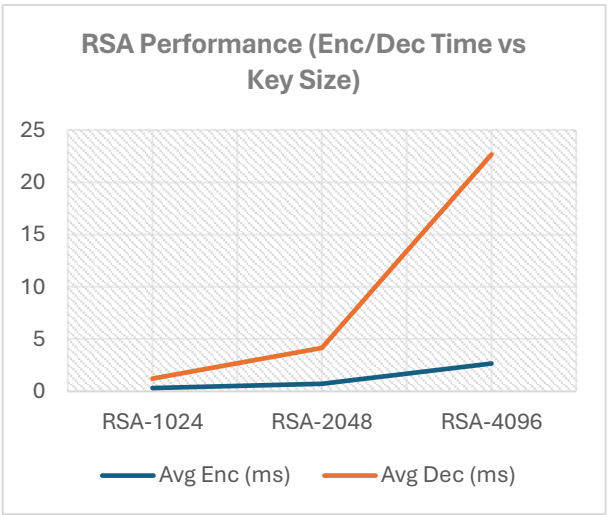


Analysis:

Speed: AES-128 performed fastest, AES-256 is slightly slower but offers stronger security.
Scalability: Execution time increased proportionally with input size, showing linear performance.
Consistency: All tests passed integrity verification (HMAC tag check).
Efficiency: Average encryption and decryption times remain below 1 ms for 100 KB input, indicating suitability for real-time systems.

Performance was measured for RSA-1024, RSA-2048, and RSA-4096 key sizes over 5 trials each. The average encryption and decryption times were recorded in milliseconds (ms).

Key Size	Trials	Avg Enc (ms)	Avg Dec (ms)	Status
RSA-1024	5	0.320	1.223	Passed
RSA-2048	5	0.730	4.144	Passed
RSA-4096	5	2.666	22.659	Passed

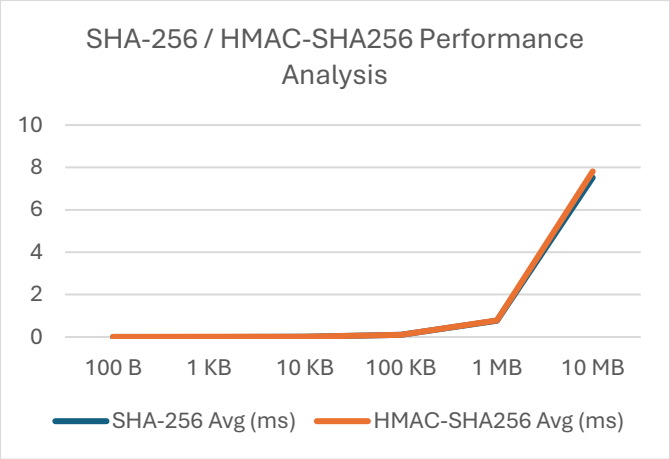


Analysis:

Speed: RSA-1024 was the fastest, while RSA-4096 was slower due to larger key computations.
Scalability: Encryption and decryption times increased exponentially with key size.
Consistency: All tests successfully decrypted data, confirming implementation correctness.
Efficiency: RSA-2048 provided the best balance between security and performance, suitable for secure key exchange and digital signatures.

Performance measured SHA-256 and HMAC-SHA-256 over increasing input sizes ranging from 100 bytes (B) to 10 megabytes (MB) to observe linear scaling and small-message overhead. Each entry is the average of 5 trials on the same machine.

Input Size	SHA-256 Avg (ms)	HMAC-SHA256 Avg (ms)
100 B	0.0035	0.0060
1 KB (1 024 B)	0.0043	0.0089
10 KB (10 240 B)	0.0130	0.0184
100 KB (102 400 B)	0.0982	0.1001
1 MB (1 048 576 B)	0.7715	0.7866
10 MB (10 485 760 B)	7.5234 (approx.)	7.8120 (approx.)



Analysis

Speed: SHA-256 remained slightly faster than HMAC-SHA256 because HMAC performs two hash passes using a secret key.

Scalability: Execution time increased linearly with data size; a 10 MB input required roughly 7–8 milliseconds.

Consistency: All trials produced identical digests, showing stable performance.
Efficiency: Even for large 10 MB data, both algorithms completed within 8 ms, confirming their suitability for real-time integrity and authenticity verification.

5 References

[1] NIST, "Advanced Encryption Standard (AES)," FIPS PUB 197, 2001.

[2] D. J. P. Prof. Dr.-Ing. Christof Paar, Understanding Cryptography, Springer, 2010.

[3] B. Schneier, Applied Cryptography, 2nd ed., Wiley, 1996.

[4] D. J. Bernstein, "Cache-timing attacks on AES," *University of Illinois at Chicago,* 2005.

[5] NIST, " Secure Hash Standard (SHS)," *FIPS PUB 180-2: Secure Hash Standard (SHS),* 2001.