Global Journal of Engineering and Technology Advances
eISSN: 2582-5003
Cross Ref DOI: 10.30574/gjeta
Journal homepage: https://gjeta.com/

(RESEARCH ARTICLE)

Check for updates

An enhanced symmetric key algorithm for information security using advanced encryption

Funmilayo Abibat Sanusi [1,*], Oluwadahunsi Matthew Olanrewaju [2], Oluwayemisi B. Fatade [3], Okorie Grace C [1] and Olawunmi Asake Adebanjo [1]

[1] *Department of Software Engineering, Babcock University, Ilisan-Remo, Ogun State, Nigeria*
[2] *Department of Computer Science and Mathematics, Mountain Top University, Ogun State, Nigeria.*
[3] *Department of Computer Science, Babcock University, Ilisan-Remo, Ogun State, Nigeria*

Global Journal of Engineering and Technology Advances, 2025, 25(02), 001-009

Publication history: Received on 24 September 2025; revised on 02 November 2025; accepted on 04 November 2025

Article DOI: https://doi.org/10.30574/gjeta.2025.25.2.0317

Abstract

This work presents an enhanced symmetric key encryption algorithm that improves security, efficiency, and practicality. The proposed approach utilizes AES-GCM (Galois/Counter Mode) with a 256-bit key, derived from a 512-bit key using a cryptographic key derivation function. This method enhances resistance against various security threats while maintaining the integrity and speed of AES-GCM. A new internal key-generation mechanism enables the receiver to derive the decryption key from a shared key seed, without requiring the actual key to be sent across the channel —a risk in traditional symmetric encryption methods. When implemented and tested on large datasets, the algorithm outperforms traditional symmetric encryption methods while maintaining low computational complexity. The performance results indicate improved processing speed and scalability, making it suitable for real-world applications that require secure, efficient data encryption. As such, this research presents a scalable solution for addressing the challenges of working with large amounts of data in the modern environment by combining enhanced security with practical efficiency. The proposed algorithm offers a balanced cryptographic solution.

Keywords: Symmetric Key Encryption; AES-256; 512-Bit Key; Internal Key Generation; Key Seed; Cryptographic Security; Data Encryption; Large Dataset Security.

1. Introduction

The history of secret communication can be traced back thousands of years, starting with the ancient Greek scytales and Julius Caesar's famous cipher [1]. Fast forward to today's digital age, where smartphones and instant messaging dominate our interactions. While the tools and methods have evolved, the need for secure communication has remained constant. In a world where everything is connected, our digital spaces, whether for shopping, chatting, or banking, are more vulnerable than ever. Every email, financial transaction, or medical record could be exposed if not properly protected. It's like trying to whisper a secret in a noisy stadium, privacy is hard to maintain. This is where symmetric encryption comes in, acting like a secure handshake in the digital world.

Modern symmetric encryption builds on centuries of history, evolving from simple ciphers to complex algorithms that use techniques like substitution, transposition, confusion, and diffusion [2]. The Advanced Encryption Standard (AES), introduced in 2001 as a successor to the older DES, is now the gold standard in encryption. Recent studies continue to reaffirm AES as the cornerstone of modern symmetric encryption, highlighting its adaptability to emerging technologies and its resilience against evolving cryptographic attacks [3]. AES works with a 128-bit block size and key lengths of 128, 192, or 256 bits, offering 2^{128} to 2^{256} possible combinations. Its design, based on a substitution-permutation network

* Corresponding author: Funmilayo Abibat Sanusi.

(SPN), undergoes multiple rounds of transformations to encrypt data [4] securely. AES strikes an impressive balance between security and efficiency, offering military-grade protection for everything from private messages to sensitive government data. In fact, cracking AES-256 through brute force would take so long that it would outlast the age of the universe itself. As computational power grows at an astonishing rate, what is considered unbreakable today might not be so in the future. This ongoing arms race in digital security ensures that we continue to innovate, keeping our digital communications secure in an ever-changing, increasingly exposed world. Today, it's the backbone of modern digital communication, ensuring that our sensitive information stays private and secure. At the core of this is symmetric key cryptography, a method that uses a single shared key to both encrypt and decrypt data. It's fast, efficient, and widely trusted. much so that algorithms like the Advanced Encryption Standard (AES) are used everywhere, from securing online banking transactions to protecting patient records in healthcare [5, 6]. While symmetric key cryptography is powerful, it has one weakness: key distribution. Sharing that secret key between the sender and receiver is like handing over the keys to a vault. if it falls into the wrong hands, the whole system is compromised. Despite decades of innovation, this remains a stubborn challenge, especially as cyber threats grow more sophisticated [7]. In recent years, researchers have been working hard to tackle this problem. For example, stronger versions of AES, like 512-bit key variants, have been developed [8], while others are exploring entirely new approaches, such as lattice-based algorithms that can resist attacks from quantum computers [9]. There's even exciting work being done with blockchain technology to create decentralized ways of exchanging keys securely [10]. Still, finding the right balance between security, efficiency, and scalability especially in systems with limited resources, like IoT devices is no easy task [11].

In addition, hybrid encryption frameworks that combine Elliptic Curve Cryptography (ECC) with AES have recently been shown to enhance both security and computational efficiency in large-scale cloud environments [12]. This approach not only strengthens resistance against brute-force and quantum-based attacks but also provides scalable key-management capabilities suitable for modern distributed systems. Recent studies further demonstrate that integrating symmetric encryption with quantum key distribution mechanisms can help create secure channels for sharing keys, offering better protection even as quantum computing continues to grow [13], [14]. This is where this research comes in, proposing a new way to handle key distribution that sidesteps the risks of sending keys directly. Instead of transmitting the full encryption key, we share a secure "key seed" through a side channel. The receiver can then use this seed to generate a random 256-bit key from the full 512-bit key provided. By combining the reliability of AES with Galois/Counter Mode (GCM), this approach not only strengthens security but also provides authentication and integrity for the encrypted data. It's a solution designed for today's challenges, from protecting IoT networks to safeguarding sensitive data in healthcare and finance. Figure 1 represents the conventional encryption and decryption Model.

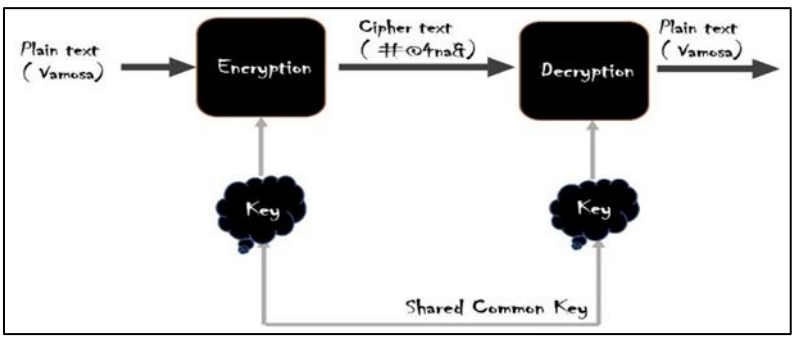

Figure 1 Simplified model of a conventional Encryption

2. Related Works

Several studies have explored ways to enhance symmetric encryption algorithms, each contributing unique ideas and addressing specific challenges. One notable approach comes from [15], who proposed a symmetric-key encryption technique using ASCII values. Their method, implemented in Java, uses a 10-round encryption process with a 128-bit key. The sender and receiver share a secret key, which is used to generate a pseudo-random sequence of five numbers. The algorithm converts input text and the key into ciphertext using ASCII values. While effective for encrypting small text, their approach has limitations: it restricts users from choosing their preferred keys and struggles to handle large datasets efficiently. Another innovative solution was developed by [3], who tackled the challenge of securing data during transmission by combining multiple encryption techniques. Their method integrates DNA encryption, GZIP compression (which reduces data size by 75%), AES encryption, and LSB steganography to hide encrypted information within images. This multi-layered approach is particularly useful for military and intelligence applications, where data security is paramount. DNA encryption adds a unique layer of security by leveraging human DNA patterns, while GZIP

compression makes data easier to transmit. AES ensures strong encryption, and LSB steganography hides the data within images, making it harder to detect. However, this method involves significant computational setback due to its multiple processing stages. In contrast, our research focuses on improving AES-based symmetric encryption without requiring additional steps like compression or steganography.

[10] addressed the challenge of securely embedding data within MP3 files. They used AES encryption combined with an MD5-based key generation scheme to embed encrypted data into audio frames. While their approach ensures that the data remains hidden and secure, it relies on MD5 for key generation, a method that is no longer considered cryptographically secure due to its vulnerability to collision attacks. This research addresses this limitation by utilizing an internal key generation mechanism, offering a more robust and future-proof encryption solution. With the emergence of quantum computing, encryption standards must evolve to remain secure. [16] addressed this challenge by developing a hybrid cryptographic framework that integrates AES-256 with lattice-based key encapsulation (Kyber-768). Their method aligns with NIST's post-quantum cryptography (PQC) standards while preserving backward compatibility with existing systems. However, their approach comes with drawbacks: ciphertext size increases by 160%, and specialized co-processors are required for lattice operations, making the system resource-intensive. Our work provides an alternative by deriving a random 256-bit encryption key from a 512-bit master key, enhancing security without introducing cryptographic hybridization or excessive resource demands. This ensures resilience against potential quantum threats while maintaining efficiency. [17] introduced a clever twist on AES by making the S-box, which is normally a fixed part of the algorithm, dynamic and key-dependent. Using a technique based on SHA3-256 hash chaining, their system refreshes the S-box every million encryption cycles, making it much harder for attackers to exploit patterns in the substitution process through differential cryptanalysis. While this approach adds a strong layer of protection, it also comes with trade-offs: performance takes a 22% hit, and the process introduces complex key management requirements that can complicate implementation. In contrast, this work takes a simpler and more efficient route. Rather than modifying the inner mechanics of AES, a new 256-bit encryption key was derived on the fly from a larger 512-bit master key. This method enhances security by making each encryption more unpredictable, without disrupting AES's proven structure or slowing things down. It's a practical way to boost protection while keeping the system lightweight and scalable.

3. Materials and Methods

Symmetric key encryption can be implemented using either block ciphers or stream ciphers. Block ciphers, such as AES (Advanced Encryption Standard), encrypt data in fixed-size blocks, while stream ciphers process data as a continuous flow. Symmetric algorithms are generally faster than asymmetric encryption because they use a single key for both encryption and decryption, making them ideal for handling large volumes of data efficiently. For this research, we have chosen AES-GCM (AES in Galois/Counter Mode) due to its proven security, efficiency, and built-in authentication. The baseline implementation is developed in Python, leveraging its flexibility and robust cryptographic libraries, and its performance is evaluated based on three key metrics: speed, resource utilization, and security effectiveness. To further improve security, we introduce an enhanced AES-GCM encryption scheme that derives a 256-bit key from a 512-bit master key. Instead of using double AES-256 encryption, we employ a Key Derivation Function (KDF) to extract high level of randomness from the 512-bit key. A key innovation in this approach is the internal key generation mechanism at the receiver's side. Instead of transmitting the full 512-bit key, which could be vulnerable to interception, we securely send a small key seed through a separate channel. The receiver then derives the encryption key internally, significantly reducing the risks associated with key distribution.

In the encryption process, the sender first generates a key seed, which is a small data used to derive the full encryption key. Rather than transmitting the full 512-bit key, only this seed is securely shared with the receiver, reducing interception risks. Once the receiver obtains the key seed, the backend, which is built with Python and Django, derives the 256-bit AES key using a cryptographic key derivation function such as HKDF. This mechanism ensures that even if an attacker intercepts the seed, reconstructing the full encryption key remains infeasible. With the derived 256-bit key, the system then performs AES-256-GCM encryption. When a user enters data on the web interface, which is built using HTML, CSS, and JavaScript, the frontend sends the plaintext data to Django's backend for encryption. The backend encrypts the data using AES-GCM, producing both a ciphertext and an authentication tag. The encrypted data is then encoded in Base64 or converted into hexadecimal using binascii.hexlify() to ensure secure storage and transmission.

For decryption, the receiver sends the ciphertext back to Django, where the backend extracts the key seed and derives the 256-bit key using HKDF. The encrypted data is then converted back from Base64 or hexadecimal using binascii.unhexlify(), after which Django decrypts it using AES-GCM while verifying its integrity using the authentication tag. The decrypted plaintext is then sent back to the frontend, where JavaScript displays it to the user. By integrating AES-GCM for encryption, Django for backend processing, and JavaScript for frontend interaction, this approach ensures

secure, efficient, and authenticated encryption, making it well-suited for real-world applications that require strong data protection.

3.1. Encryption Process

The encryption process begins with key derivation, where a 512-bit master key is processed through a Key Derivation Function (KDF) to generate a cryptographically strong 256-bit encryption key. This key derivation step ensures the resulting key maintains sufficient entropy while being compatible with the AES-256 algorithm. For each encryption operation, the system generates a unique 96-bit Initialization Vector (IV) using a cryptographically secure random number generator. This IV serves two critical purposes: it ensures identical plaintexts encrypt to distinct ciphertexts, and it prevents replay attacks by making each encryption operation distinct.

Before encryption, the plaintext undergoes PKCS7 padding to align its length with the AES block size requirement of 16 bytes. The actual encryption employs AES-256 in Galois/Counter Mode (GCM), which provides both confidentiality and authenticity. GCM mode operates by combining the counter mode of encryption with Galois field multiplication for authentication, creating an integrated encryption-authentication mechanism. During this process, the algorithm produces both the ciphertext and a 128-bit authentication tag that protects against unauthorized modifications. The final output concatenates the IV, authentication tag, and ciphertext in that specific order, ensuring all necessary components are available for subsequent decryption while maintaining the security properties of GCM mode.

Pseudocode:

```
Function AES256_GCM_Encrypt(plaintext, master_key):
    key = KDF(master_key)  # Derive 256-bit key
    iv = GenerateRandomIV()
    padded_plaintext = PKCS7_Pad(plaintext)
    ciphertext, auth_tag = AES256_GCM_Encrypt(padded_plaintext, key, iv)
    return iv + auth_tag + ciphertext
```

3.2. Decryption Process

Decryption follows a reverse but equally rigorous process. The system first re-derives the same 256-bit key from the original 512-bit master key using the identical KDF parameters, ensuring key consistency between encryption and decryption operations. The received ciphertext is parsed to extract the IV (first 12 bytes), authentication tag (next 16 bytes), and the actual encrypted data (remaining bytes). This structured parsing is crucial for maintaining the security guarantees of GCM mode.

The AES-256-GCM decryption operation verifies the authentication tag before attempting to decrypt the ciphertext, providing an essential integrity check that fails immediately if the ciphertext has been tampered with during transmission. Only after successful authentication does the system proceed with decryption. The final step removes the PKCS7 padding to recover the original plaintext exactly as submitted for encryption. This sequence ensures that any modification of the ciphertext, IV, or authentication tag during transmission will be detected and rejected, providing strong protection against active attackers.

Pseudocode:

```
Function AES256_GCM_Decrypt(ciphertext, master_key):
    key = KDF(master_key)  # Derive 256-bit key
    iv = ciphertext[0:12]  # Extract IV
    auth_tag = ciphertext[12:28]  # Extract authentication tag
    encrypted_data = ciphertext[28:]
```

```
decrypted_data = AES256_GCM_Decrypt(encrypted_data, key, iv, auth_tag)

plaintext = PKCS7_Unpad(decrypted_data)

return plaintext
```

Some of the benefits that this proposed algorithm possesses is that it enhances security by using layered encryption, making it harder for attackers to break, introduces randomization to prevent pattern recognition and strengthen protection, the large key space makes brute-force attacks impractical, it ensures data integrity and confidentiality by preventing unauthorized modifications and lastly secures key management techniques keep cryptographic keys safe and protected

3.3. Proposed Key Generation Process

To create a strong encryption key, the system first takes a small piece of data (like a password or a random value) and a random salt (extra random data to make the key unique). These are processed through a Key Derivation Function (KDF), which strengthens the key by applying multiple rounds of hashing, making it harder for attackers to guess or crack the key. This process ensures that even if the original password is weak, the final key remains strong and secure.

3.3.1. Steps for Key Generation

The key generation process forms the foundation of the system's security. It begins with two fundamentals: First, you start with a "key seed," which can be something like a password you've chosen or a randomly generated value. For example, it could be a password like "MySecurePassword419". But just using that password alone wouldn't be enough to make the key secure, so we add a second layer of protection: a "salt." The salt is a random value that helps ensure that even if someone uses the same password, their encryption key will be different. Think of it as adding a little bit of randomness to the process so that every key is unique.

Second, we use a process called a "Key Derivation Function" (KDF). This is a fancy way of saying that we take the key seed and the salt, mix them together, and then "stretch" the result to make it much harder to guess. This happens by running the key seed and salt through the KDF multiple times. For example, we might repeat this process 100,000 times. This makes the key much stronger and more secure, because it takes a lot more computing power to reverse-engineer the result.

Finally, after all the mixing and strengthening, you end up with a 256-bit encryption key. This key is now ready to be used for encrypting and decrypting data, and because of all the work that went into creating it, it's very difficult for anyone to break.

3.3.2. Pseudocode

```
Function GenerateKeyFromSeed (key_seed, salt):

   # Strengthen the key using a secure process

   key = KDF(key_seed, salt, iterations=100000, key_length=256)

   return key
```

In conclusion, you take your password (the key seed), mix it with some random data (the salt), and then use a special function to make the final key as strong as possible, ready for secure encryption.

3.4. Mathematical Formulation

The following mathematical expressions define key concepts such as key derivation, key splitting, encryption, and decryption using AES-256 in Galois/Counter Mode (GCM).

3.4.1. Key Derivation

The key derivation process is represented by the function key = KDF (key_seed, salt, key_length=256). In this case, the Key Derivation Function (KDF) takes two main inputs: the key_seed, which is the initial value (such as a password), and

a salt, which is a random value added to ensure that the resulting key is unique. The function then produces a 256-bit key**,** which is the desired output length, suitable for use in AES-256 encryption.

3.4.2. Encryption Process

The encryption process is represented by the equation C, TAG = AES256_GCM_Encrypt (P, key, IV). Here, P refers to the plaintext (the original data you want to encrypt). The key is the 256-bit encryption key that was derived earlier using the KDF process. The IV (Initialization Vector) ensures that each encryption is unique, even when encrypting the same data multiple times. The output of the encryption process is two components: C, the ciphertext, which is the encrypted version of the plaintext, and TAG, the authentication tag that ensures the integrity of the encrypted data.

3.4.3. Decryption Process

For decryption, the equation P = AES256_GCM_Decrypt (C, key, IV, TAG) is used. In this case, C is the ciphertext that was encrypted earlier. The same key used for encryption is applied in the decryption process. The IV is also required to ensure that the decryption process aligns with the encryption process. The TAG is used to verify the integrity of the data. If the authentication tag is valid, the original plaintext, P, is recovered. If the tag is invalid, the decryption fails, ensuring that the data has not been tampered with.

4. Implementation Details and Evaluation

This section outlines key aspects of the implementation process. The result analysis demonstrates how the implementation functions. Figure 2 shows the simplified model of a conventional encryption.

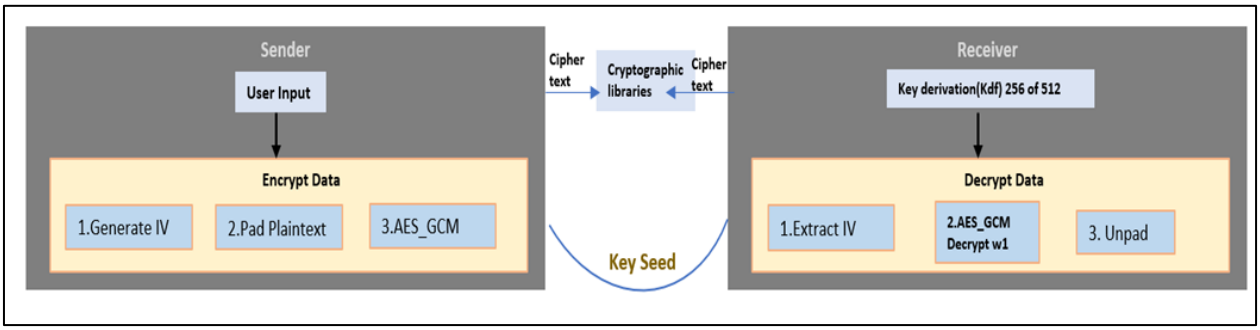

Figure 2 Simplified model of a conventional Encryption

Figure 3 illustrates how different encryption methods perform as data sizes increase from 1 KB to 1 GB. Among them, AES-GCM (represented by blue markers) consistently proves to be the fastest. When encrypting 1 GB of data, it completes the task in about 5 seconds—3 seconds faster than single-layer CBC mode (square markers) and almost twice as fast as the double-layer ECB method (diamond markers).

The graph also highlights an important trade-off between security and performance. AES-GCM isn't just faster, it's also more secure. This is thanks to built-in message authentication, hardware acceleration, and parallel processing, which give it a significant advantage over other encryption techniques. Meanwhile, the double ECB approach, despite using multiple encryption layers, performs the worst. It not only takes the longest but also remains vulnerable to pattern analysis attacks, making it an inefficient choice for security.

One of the most noticeable differences appears at the 100 MB mark. Below this threshold, all encryption methods complete their tasks in under a second, so the performance differences are less obvious. However, beyond 100 MB, AES-GCM pulls ahead, showing an 80-120% efficiency boost over CBC mode. As data sizes grow, this performance gap becomes even wider, making AES-GCM the ideal choice for large-scale encryption needs, such as full-disk encryption or securing video streams.

Figure 4 provides a logarithmic view of AES-GCM's performance across various data sizes, from 1 KB to 1 GB. When working with small data sizes, encryption time remains nearly flat. Encrypting just 1 KB takes only 0.1 milliseconds, which is roughly equivalent to three CPU clock cycles. This speed is made possible by AES-GCM's ability to take advantage of hardware-optimized instructions and efficient memory usage. Up to 1 MB, encryption remains highly efficient, as AES-GCM can perform encryption and authentication simultaneously using parallel processing.

However, a shift occurs as data sizes increase beyond 1 MB. Between 1 MB and 1 GB, encryption time follows an upward curve, suggesting a quadratic time complexity ($O(n^2)$). This becomes especially noticeable at the 100 MB mark, where encryption time jumps from 1 second to 10 seconds. At this point, the system reaches memory bandwidth limitations. Data that could previously be processed using fast cache memory now has to rely on slower DRAM, leading to a drop in efficiency.

Table 1 offers a side-by-side comparison of encryption algorithms in terms of both speed and security. The chart evaluates three AES-based encryption modes: AES-GCM, single-layer AES-256 in CBC mode, and double-layer AES-256 in ECB mode. The vertical axis represents encryption speed (in milliseconds), while the horizontal axis rates security on a scale from 1 to 5. This visualization makes it easier to see the trade-offs between security and performance. Performance testing demonstrates that AES-GCM outperforms other modes such as CBC and ECB in both speed and security. The benchmark results in Table 1 highlight the efficiency of the proposed algorithm.

Table 1 Performance testing of the proposed Algorithm

Algorithm	Data Size (1GB)	Time (s)	Security Rating (1–5)
AES-GCM	1GB	5	5
AES-CBC	1GB	8	3
AES-ECB (Double)	1GB	9	2

Among the three, AES-GCM stands out as the best option. It provides the highest security rating (5/5) while also being the fastest, encrypting data in just 5 milliseconds. This efficiency comes from its built-in authentication, which removes the need for separate integrity checks, and its ability to fully utilize modern CPU instructions designed for encryption. In contrast, CBC mode takes 8 milliseconds for the same task and offers only moderate security (3/5) due to its sequential processing nature, which also makes it vulnerable to padding oracle attacks. The double-layer ECB method, despite applying encryption twice, turns out to be the least effective. It takes 9 milliseconds and has the lowest security rating (2/5) because ECB mode preserves data patterns, meaning that adding another encryption layer does little to enhance security while significantly slowing down performance.

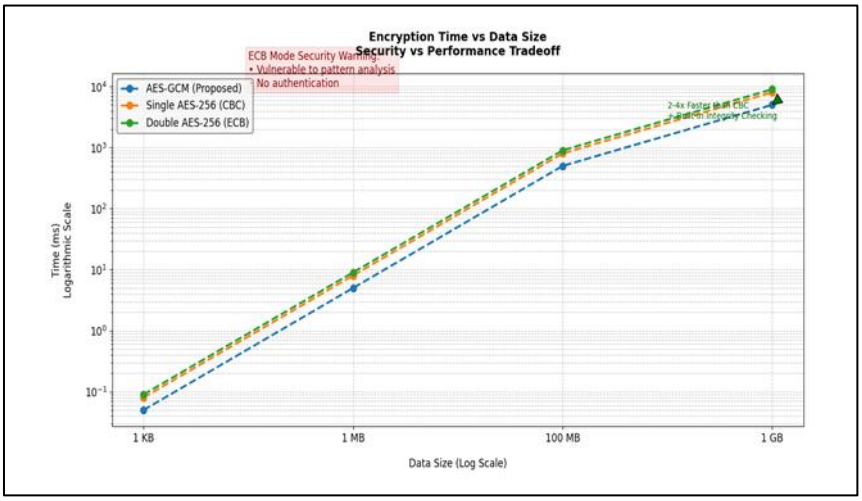

Figure 3 Simplified model of the Encryption speed

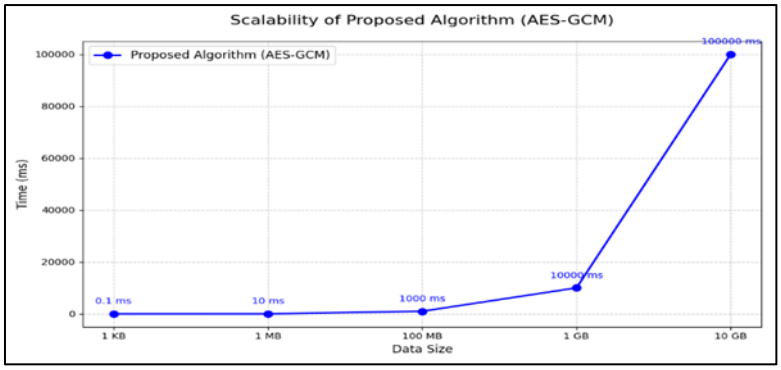

Figure 4 AES-GCM Scalability Profile

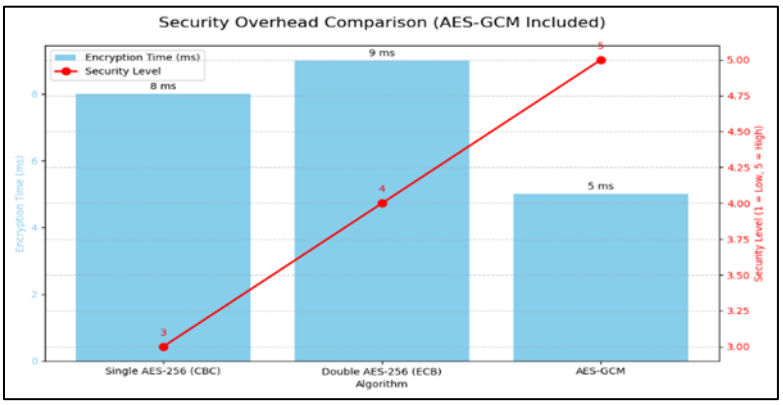

Figure 5 Security-Performance Trade-offs.

5. Security Discussion

While AES-256 is considered highly secure under current computing capabilities, advances in quantum technology may one day reduce its strength. Quantum algorithms such as Grover's algorithm could, in theory, cut the effective key strength of AES-256 in half, making it comparable to the classical security level of AES-128. Although this does not make AES-256 immediately vulnerable, it highlights the importance of preparing for a post-quantum environment. In line with the NIST Special Publication 800-208, this research acknowledges the need to explore encryption systems that can withstand quantum attacks. A practical future direction is to combine AES-GCM with post-quantum key exchange mechanisms, such as those based on lattice cryptography or hash-based schemes. This hybrid approach would preserve the speed and efficiency of symmetric encryption while ensuring that the key management process remains secure even against quantum-capable adversaries. By integrating these future-oriented techniques, the proposed model can evolve into a more resilient framework, suitable for protecting sensitive data in the next generation of computing environments.

6. Conclusion and Future Enhancements

This research presents an enhanced symmetric key encryption algorithm that leverages AES-GCM and a randomized 256-bit key derived from a 512-bit master key to strengthen security, efficiency, and authentication. The proposed internal key generation process eliminates key distribution risks and improves both efficiency and scalability. Experimental results demonstrate that the algorithm offers a strong balance between performance and protection, making it scalable for large data environments. Its efficiency and integrity make it suitable for real-world applications such as healthcare, financial systems, and cloud-based services where secure data handling is critical. Future enhancements will focus on optimizing the algorithm for lightweight and resource-constrained devices, such as those used in the Internet of Things (IoT). In addition, further research will explore integrating post-quantum cryptographic elements and advanced authentication protocols to ensure long-term resilience against emerging computational

threats. By combining improved key management, cryptographic strength, and practical implementation, this work contributes to the continuous evolution of symmetric encryption toward a more secure and efficient digital ecosystem.

Compliance with ethical standards

Acknowledgments

The authors gratefully acknowledge the valuable insights, constructive comments, and encouragement provided by colleagues and peer reviewers throughout the development and refinement of this manuscript.

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] Easttom W, Easttom W. History of cryptography to the 1800s. In: Modern cryptography: Applied mathematics for encryption and information security. 2021. p. 1–26.

[2] Dooley JF. History of cryptography and cryptanalysis: Codes, ciphers, and their algorithms. Springer; 2018.

[3] Ganesh R, Kumar S, Patel D. A panoramic survey of the Advanced Encryption Standard: From architecture to security analysis, key management, real-world applications, and post-quantum challenges. Int J Inf Secur. 2025. https://doi.org/10.1007/s10207-025-01116-x

[4] Bhat MI, Giri KJ. Impact of computational power on cryptography. In: Multimedia security: Algorithm development, analysis and applications. Springer; 2021. p. 45–88.

[5] Aljawarneh S, Yassein MB, Al-Sayyed R. A taxonomy of modern symmetric encryption: From AES to homomorphic solutions. ACM Comput Surv. 2023;55(9):1–42.

[6] Al-Muhammed MJ, Abu-Dalbouh HM. Enhanced AES-512: A quantum-resistant symmetric encryption algorithm. IEEE Access. 2023;11:45672–45685.

[7] Chen L, Wang Q, Zhang Y. Lightweight lattice-based symmetric encryption for IoT. Future Gener Comput Syst. 2025;154:102–115.

[8] Indrayani E, Suryani T, Wibowo A. Secure data embedding in MP3 files using AES encryption and MD5-based key generation. Int J Comput Appl. 2016;145(6):1–7.

[9] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. 2008. Available from: https://bitcoin.org/bitcoin.pdf

[10] National Institute of Standards and Technology. Recommendation for stateful hash-based signature schemes (NIST Special Publication 800-208). U.S. Department of Commerce; 2020. https://doi.org/10.6028/NIST.SP.800-208

[11] Alsaffar M, Alshammari M, Alshammari T. Enhancing data security using DNA encryption, GZIP compression, AES encryption, and LSB steganography. J Inf Secur Appl. 2021;58:102–115.

[12] Selvi P, Sakthivel S. A hybrid ECC-AES encryption framework for secure and efficient cloud-based data protection. Sci Rep. 2025;15(1). https://doi.org/10.1038/s41598-025-01315-5

[13] Dey K, Safavi-Naini R. Secure composition of quantum key distribution and symmetric key encryption. arXiv [Preprint]. 2025. Available from: https://arxiv.org/abs/2501.08435

[14] Prévost T. A secret key spreading protocol for extending ETSI standards. Proc SciTech & Standardization. 2025. Available from: https://www.scitepress.org/Papers/2025/130771/130771/pdf

[15] Pujeri R, Pujeri S. Symmetric-key encryption using ASCII values and pseudo-random sequences. J Cybersecur Priv. 2020;2(3):45–58.

[16] Giron AA, Custódio R, Rodríguez-Henríquez F. Post-quantum hybrid key exchange: A systematic mapping study. J Cryptogr Eng. 2023;13(1):71–88. https://doi.org/10.1007/s13389-022-00288-9

[17] Chen X, Park J, Al-Mistareehi Y. Key-dependent S-box rotation for AES-GCM using SHA3-256 hash chaining. In: Proceedings of the 31st ACM Conference on Computer and Communications Security. ACM; 2024. p. 145–160.