Performance Evaluation and the Impact of File Size on Various AES Encryption Modes

Galih Putra Riatma[1], Bagas Satya Dian Nugraha[2], Anugrah Nur Rahmanto[3], Fitri[4]

[1]Digital Telecommunication Network Study Program, Department of Electrical Engineering, State Polytechnic of Malang, 65141, Indonesia.

[2,3]Department of Informatics, State Polytechnic of Malang, 65141, Indonesia.

[4]Department of Electrical Engineering, State Polytechnic of Malang, 65141, Indonesia.

[1]griatma@polinema.ac.id , [2]bagassatya@polinema.ac.id, [3]anugrahnur@polinema.ac.id, [4]fitri@polinema.ac.id

Abstract— **Important factors in the system are performance and information security. A secure system does not necessarily have fast performance because it takes time for encryption processing that takes time. For that, the system must use an encryption mode that suits the needs. This study measures the encryption performance of five AES methods, namely AES-ECB, AES-SIV, AES-CBC, AES-EAX, and AES-GCM on text data, and images with sizes of 1KB, 10KB, 100KB, and 1000KB. Performance testing is carried out using the same hardware and software to ensure consistency. From the analysis results, it was found that the AES-ECB (Electronic Codebook) encryption results had the fastest encryption time but sacrificed security because of the data patterns seen in the ciphertext. Meanwhile, AES-SIV (Synthetic Initialization Vector) produced performance that tended to be constant for all file sizes, without sacrificing security against nonce reuse. AES-CBC (Cipher Block Chaining) produced a time that increased as the file size increased. The larger the encrypted file, the slower the CBC encryption performance due to the chaining nature of CBC encryption. Meanwhile, EAX and GCM show significant time improvements for small file sizes but not too significant improvements for large files. From the results of EAX and GCM, it can be concluded that both modes are efficient for encrypting large files. From the analysis results, it was found that GCM mode provides strong security without a significant impact on system performance. This research can help developers when developing systems that require encryption in environments with limited resources such as embedded systems or IoT devices.**

Keywords—AES, comparison, encryption, performance, security

I. INTRODUCTION

Encryption is a fundamental technique for securing digital communication, transforming human-readable information into an unreadable format to protect confidentiality, integrity, and authentication. The Advanced Encryption Standard (AES), a symmetric block cipher standardized by the National Institute of Standards and Technology (NIST) in 2001, remains one of the most widely adopted encryption algorithms across various applications [1][2]. AES supports three key lengths—128-bit, 192-bit, and 256-bit—along with multiple encryption modes that enhance security and operational efficiency. Among these, Electronic Codebook (ECB), Cipher Block Chaining (CBC), Encrypt-then-Authenticate-then-Translate (EAX), Galois Counter Mode (GCM), and Synthetic Initialization Vector (SIV) offer distinct trade-offs in terms of security, complexity, and computational speed [3].

Despite the broad adoption of AES, there remains a lack of comparative studies examining the performance of AES modes in constrained environments, particularly on resource-limited microcontrollers. While prior research has explored encryption efficiency in general computing environments, few studies have systematically analyzed how different AES modes behave under computational constraints. Microcontrollers play a vital role in embedded systems and Internet of Things (IoT) applications, often operating with limited processing power, memory, and energy resources. In such environments, an encryption algorithm must strike a careful balance between security and performance—ensuring strong data protection without imposing excessive computational overhead that could degrade system responsiveness.

Encryption algorithms designed for standard computing platforms may not be optimized for resource-restricted devices. A poorly chosen encryption method could significantly impact system efficiency, leading to higher energy consumption, reduced operational lifespan, and slower execution times. For battery-powered devices and IoT deployments, excessive computation demands can drastically limit usability, forcing developers to make trade-offs between security and system efficiency. Selecting an appropriate encryption mode is essential for applications where real-time processing speed and minimal resource usage are critical, such as sensor networks, low-power embedded controllers, and wireless communication systems.

This research seeks to bridge that gap by evaluating the encryption and decryption performance of five AES modes across various file sizes. By identifying the most efficient mode, this study lays the groundwork for future implementation on microcontrollers, where processing power, memory, and energy consumption are critical concerns. A well-optimized encryption scheme can extend device longevity, improve system reliability, and ensure seamless data transmission, making encryption practical even in constrained environments.

Additionally, encryption plays a critical role in securing Long Range (LoRa) communication, an open, unencrypted protocol designed for long-distance, low-power data transmission [6][7]. LoRa-enabled devices are commonly used in IoT deployments, remote sensing, and smart infrastructure, where computational power is highly constrained. Given LoRa's inherent lack of built-in encryption, selecting an optimal AES mode for constrained environments becomes even more relevant. If encryption is inefficient, it could hinder LoRa's low-power advantage, limiting its usability in applications that require minimal energy consumption, such as environmental monitoring and industrial automation. By determining which AES mode offers the best balance of security and efficiency, this study contributes to enhancing LoRa-enabled device security while ensuring minimal impact on system performance.

This comparative analysis aims to provide valuable insights for developers, network administrators, and embedded system engineers, offering a practical foundation for selecting the most suitable encryption approach for real-world applications where computational resources are limited. By understanding the performance characteristics of various AES modes, future research can focus on implementing the top-performing encryption methods in embedded systems, ensuring both security and operational efficiency in constrained environments.

II. METHOD

A. System Description

This study proposes the development of a system capable of performing data encryption and decryption using five operational modes of the Advanced Encryption Standard (AES) algorithm: Cipher Block Chaining (CBC), Encrypt-then-Authenticate-then-Translate (EAX), Electronic Codebook (ECB), Galois/Counter Mode (GCM), and Synthetic Initialization Vector (SIV). The primary objective of this research is to analyze the encryption and decryption processing times across these five AES modes with varying data sizes and encryption keys [9]. The findings are expected to provide insights into the performance and security characteristics of each AES operational mode.

The hardware utilized for this study consists of a computer equipped with an Intel i5-4670 processor, 16GB RAM, and a 512GB SATA SSD. This configuration represents a modern computational environment positioned at the mid-range level of contemporary computing systems. The use of an SSD mitigates potential I/O bottlenecks associated with traditional HDD storage, ensuring a more efficient evaluation of encryption performance. It should be noted that the experiment is done using Python environments, version 3.10, developed using Visual Code under Windows 10 Operating System. All external factors such as antivirus, background programs, and precautions against any factors that might slow the process down (overheating, drivers, background I/O process) have been minimized as much as possible.

The custom-built application for this research is developed in Python 3.10, using Visual Studio Code as the integrated development environment (IDE) under the Windows 10 Operating System. Python was chosen for its rich ecosystem and support for high-resolution timing functions, which are crucial for accurately measuring the minimal processing times involved in encryption and decryption operations. The encryption processes are implemented using the PyCryptodome library, a robust and well-supported tool for handling multiple AES modes. This library enables seamless integration of various AES operational modes while ensuring that all cryptographic operations adhere to industry standards, as shown in Fig. 1.

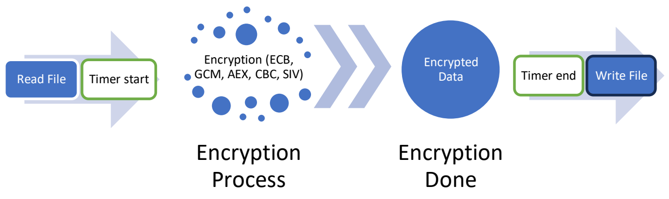

Figure 1. The flow of how the system measures time

The research adopts an experimental methodology, utilizing text data of varying sizes of 1 KB, 10 KB, 100 KB, and 1 MB. this is done to represent common usage scenarios ranging from small to large files. Randomized data ensures an even distribution, preventing encryption bias. The encryption key employed is a 256-bit symmetric key, as AES-256 is recognized for providing the highest level of security recommended by the National Institute of Standards and Technology (NIST) [10].

The encryption and decryption processes are executed using a custom-built application developed in Python [11]. To measure encryption and decryption time, the study employs the perf_counter_ns() function from Python's time module, chosen due to its high-resolution timing capability. Process isolation is maintained to ensure results are not affected by input/output (I/O) operations.

Each combination of data size and encryption key undergoes ten iterations, with the average processing time computed to minimize statistical noise caused by operating system interruptions. Time measurements focus solely on the encryption process, excluding the time required for reading and writing encrypted files. This approach effectively eliminates potential errors caused by I/O bottlenecks, ensuring accurate performance evaluation.

The overall system architecture is modular, ensuring a clear separation of concerns that makes precise performance measurement possible. This design approach echoes the modular implementations found in cryptography-based systems [12][13]. The key components include:

- Encryption Module: This module is responsible for instantiating the AES cipher for each of the five modes. Each mode is configured with its specific parameters, such as initialization vectors (IVs) for CBC, synthetic IV generation for SIV, or authentication tag computation for EAX and GCM [14]. This separation ensures that mode-specific overhead can be independently analyzed.

- Data Input Module: The system generates randomized text data covering a range of sizes: 1 KB, 10 KB, 100 KB, and 1 MB. Using similar data prevents any bias in the encryption process and simulates a variety of potential real-world scenarios, from small sensor payloads to larger document files. The similar but not exact data nature of the input ensures an even distribution of bits, eliminating issues caused by repetitive patterns [15].

- Timing Module: High-resolution timing is captured using Python's time.perf_counter_ns() function [16]. Immediately before the encryption operation begins and after it ends, timestamps are recorded to obtain precise measurements in nanoseconds. By isolating the encryption operation from file I/O processes, the timing module ensures that the measured processing times reflect only the computational aspects of the encryption techniques [17].

- Logging Module: For each experimental run, performance metrics (encryption and decryption times) are logged. Each test is repeated ten times for every combination of data size and encryption mode, and the average processing time is computed to minimize statistical noise that might arise from operating system interruptions [18].

Each component of the system is designed to facilitate a robust and repeatable evaluation of AES encryption performance [19]. By using a consistent hardware and software environment, the system minimizes variability, allowing clear attribution of performance differences to the encryption modes rather than external factors. The decision to employ a 256-bit key standardized across all experiments ensures that variations in performance are due solely to the mode of operation rather than differences in cryptographic strength.

Furthermore, while the experimental setup is executed on a moderately powerful desktop system, the insights gained from this study are targeted toward future implementations on resource-constrained microcontrollers. Especially in contexts such as LoRa communication systems. In these environments, even minor differences in computational efficiency can have significant impacts on energy consumption and overall system responsiveness.

B. Program Initialization

All necessary variables and parameters required for the encryption process are initialized. This includes defining the encryption key, preparing the data to be encrypted, and creating the corresponding encryption object. The encryption key is 8 characters alphanumeric word. Which will be the same for all AES operation modes conducted in this experiment to make sure the key does not affect performance.

```
from Crypto.Cipher import AES
import time

key = b'Sixteen byte key'
data = b'This is the data to encrypt'
cipher = AES.new(key, AES.MODE_EAX)
```

C. Timestamp Before Encryption

A precise timestamp is recorded immediately before the encryption process begins using the time.perf_counter_ns() function. This function provides high-resolution time measurement in nanoseconds, ensuring exceptional accuracy for execution time evaluation.

```
start_time = time.perf_counter_ns()
```

This statement saves the current counter value in the start_time variable. Later, after the encryption process is complete, capturing another timestamp and computing the difference allows us to determine the exact number of nanoseconds that the encryption took. By using this high-precision timer, we ensure that our performance measurements are reliable and that any variations in the encryption latency can be detected and analyzed accurately. This approach is particularly valuable when repeating the encryption process multiple times to derive an average execution time, thus providing a robust basis for comparing the different AES operational modes.

D. Implementation of AES Modes

The encryption process is executed on the preprocessed data, ensuring that only the encryption operation is performed without any additional tasks such as file reading or writing.

```
ciphertext,tag=cipher.encrypt_and_digest(data)
```

Each AES mode offers a distinct approach to how plaintext is processed, affecting the encryption speed and overall resource footprint. In our custom-built Python-based application, we leverage the PyCryptodome library to specify the mode explicitly when creating the AES cipher object. Below we describe the implementation details for each mode:

- AES-ECB (Electronic Codebook):

In ECB mode, each block of plaintext is encrypted independently, with no use of an initialization vector (IV) or chaining between blocks. This simplicity makes ECB the fastest in terms of pure computation; however, it is known to be less secure since identical plaintext blocks produce identical ciphertext blocks. In our implementation, ECB is invoked using:

```
cipher_ecb = AES.new(key, AES.MODE_ECB)
ciphertext = cipher_ecb.encrypt(pad(data, AES.block_size))
```

Note that proper padding is applied, which is critical because the plaintext length must be a multiple of the block size.

- AES-CBC (Cipher Block Chaining):

CBC mode improves security by XORing each plaintext block with the previous ciphertext block before encryption, with a randomly generated IV for the first block. This chaining introduces additional overhead, as every block's encryption depends on the output of the preceding block, and the IV must be transmitted securely alongside the ciphertext. In our implementation:

```
iv = get_random_bytes(AES.block_size)
cipher_cbc = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher_cbc.encrypt(pad(data, AES.block_size))
```

The process of generating the IV, performing the XOR operation, and handling padding explains the longer processing time observed for larger file sizes.

- AES-EAX (Encrypt-then-Authenticate-then-Translate):

EAX mode is designed to provide both confidentiality and data integrity via authentication. It combines encryption with an authentication tag generated during the `encrypt_and_digest` operation. Though it adds the overhead of computing the authentication tag, this mode is often preferred for its robustness against tampering:

```
cipher_eax = AES.new(key, AES.MODE_EAX)
ciphertext, tag = cipher_eax.encrypt_and_digest(data)
```

The inherent verification process adds to the processing time, especially noticeable with larger data sizes.

- AES-GCM (Galois/Counter Mode):

GCM is similar to EAX in that it offers authenticated encryption. However, GCM uses a counter (CTR) mode for encryption and a Galois field multiplication for authentication, which, when optimized, allows for parallel processing of blocks. In our setup, we initialize GCM as follows:

```
cipher_gcm = AES.new(key, AES.MODE_GCM)
ciphertext, tag = cipher_gcm.encrypt_and_digest(data)
```

While GCM provides excellent throughput in parallel hardware implementations, our Python implementation still reveals slight overhead due to the additional multiplication steps for authentication.

- AES-SIV (Synthetic Initialization Vector):

SIV mode is designed for nonce-misuse resistance, ensuring secure encryption even if IVs or nonces are repeated. SIV mode computes a synthetic IV using both the message and key, then uses the IV in the standard encryption process. While this design offers strong security guarantees, it incurs extra computational cost. Our implementation leverages the available support in PyCryptodome (or an alternative library if needed) with:

```
cipher_siv = AES.new(key, AES.MODE_SIV)
ciphertext, tag = cipher_siv.encrypt_and_digest(data)
```

The additional steps in synthesizing the IV result in relatively higher processing times compared with other modes.

E. Timestamp after Encryption

Immediately upon completion of the encryption process, another timestamp is recorded using time.perf_counter_ns(). This serves as the end time, which is then utilized to compute the encryption duration.

```
end_time = time.perf_counter_ns()
```

The time difference between the start and end timestamps is computed to determine the encryption duration. This measurement captures only the encryption processing time, eliminating external influences such as file reading and writing operations.

```
encryption_duration = end_time - start_time
print(f"Durasi Enkripsi: {encryption_duration} nanodetik")
```

This subtraction yields the total time taken solely for the encryption operation, expressed in nanoseconds. It is important to note that this measurement is isolated from any external I/O operations, such as reading from or writing to a file; hence, the computed duration reflects only the computational time spent on encryption. High-precision timing in this context is critical, since even minor variations can be significant when comparing different encryption modes or when optimizing for resource-constrained environments. For easier readability, the result from nanosecond will be converted to millisecond (ms).

III. RESULTS AND DISCUSSION

Table 1 presents the encryption speed measurements for various file sizes. The measurements were conducted by recording the encryption time in milliseconds (ms) for each file size across five trials, ensuring that the data variation is captured with greater accuracy. the table displays the average encryption time calculated from these five trials. This table aids in analyzing the performance of the encryption algorithm, particularly by illustrating how encryption time varies with different file sizes. The filesizes are presented in KiloByte(KB), the file type should not matter as in this operation, but for clarity, the files are mix of image and text file, as shown in Table I.

TABLE I
MEASURED TIME FOR ENCRYPTION

AES Mode	File Size (kB)	Average time (ms)
CBC	1	2.016
	10	3.9
	100	23.292
	1000	57.321
EAX	1	0.301
	10	0.546
	100	1.908
	1000	10.706
ECB	1	0.095
	10	0.108
	100	1.087
	1000	2.77
GCM	1	0.331
	10	0.474
	100	2.311
	1000	9.226
SIV	1	1.984
	10	1.797
	100	4.87
	1000	17.618

Figure 2 illustrates the comparison of AES encryption speeds for several modes of operation using file sizes ranging from 1 to 1000 KB. The graph demonstrates that AES-ECB (Electronic Codebook) exhibits a linear relationship between encryption time and file size. Although AES-GCM, AES-EAX, and AES-SIV are not as fast as AES-ECB, they display relatively stable encryption times regardless of the increasing file size. In contrast, AES-CBC shows a sharply increasing encryption time as the file size grows, indicating that this mode is less efficient when handling larger files.

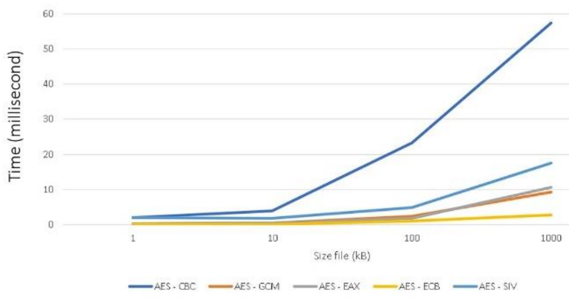

Figure 2. The chart of time needed to complete each encryption operations.

n this study, we convert the measured encryption times into a normalized performance metric, defined as the number of operations (i.e., file encryptions) performed per second (ops/s). By regarding each file encryption as a single discrete operation, we transform raw timing data into a more intuitive and scalable measure of throughput. This metric enables us to compare different AES modes on a common scale, effectively decoupling performance from the specific file sizes used during testing.

Using operations per second as a performance indicator offers several advantages. First, it provides a standardized means of evaluation that facilitates direct comparison across diverse encryption configurations. For instance, rather than stating that an AES mode operates at a specific latency in milliseconds, framing the result in ops/s conveys the system's capacity to handle large number of sequential encryptions, which is especially relevant in high-throughput environments. Second, this metric abstracts the dependency on file size by normalizing performance measurements; even when only small data packets are processed as is typical in many IoT applications the ops/s metric remains directly applicable to scenarios where high-frequency encryption transactions are required.

By converting the encryption times into a performance metric, the number of operations (i.e., encryptions) performed per second. We do this by treating a single file encryption as one operation and calculating

$$Operations\ per\ second\ = \left(\frac{1000}{time}\right) \qquad (1)$$

Using the equation 1 we get the performance metric of each file size of the fastest (ECB) and slowest (CBC) method of AES. We get the result in Table II.

TABLE II
PERFORMANCE METRIC OF FASTEST AND SLOWEST ENCRYPTION

File size (KB)	CBC Time (ms)	CBC Performance (ops/s)	EBC time (ms)	EBC Performance (ops/s)
1	2	500	0,09	11111.11
10	3,4	294.12	0.1	10000
100	23	43.48	1.87	534.76
1000	57	17.54	2.77	361.02

After getting the operation per second metric, we can calculate the performance efficiency metric using the equation 2.

$$efficiency = \left(\frac{\text{fastest}-\text{slowest}}{\text{slowest}}\right) \times 100\% \qquad (2)$$

- For 1 KB Files:
  ECB Speed = 11111.11 ops/s
  CBC Speed = 500 ops/s
  $efficiency = \left(\frac{10611.11}{11111.11}\right) \times 100\% = \ 95.50\ \%$
- For 10 KB Files:
  ECB Speed = 10000 ops/s
  CBC Speed = 294.12 ops/s
  $efficiency = \left(\frac{9705.88}{10000}\right) \times 100\% = \ 97.06\ \%$
- For 100 KB Files:
  ECB Speed = 534.76 ops/s
  CBC Speed = 43.48 ops/s
  $efficiency = \left(\frac{491.28}{534.76}\right) \times 100\% = \ 91.85\ \%$
- For 1000 KB Files:
  ECB Speed = 361.02 ops/s
  CBC Speed = 17.54 ops/s
  $efficiency = \left(\frac{343.48}{361.02}\right) \times 100\% = \ 95.14\ \%$

In short, the fastest mode, ECB only needs roughly 5 – 3% of time needed by the slowest mode CBC. Conclusions drawn from these tests indicate that AES-ECB exhibits stable encryption times even as the file size increases. However, it demonstrates significant security weaknesses, as patterns in the plaintext may become apparent in the ciphertext. This mode lacks both chaining and authentication, rendering it less secure overall. In contrast, AES-CBC processes each block of plaintext by combining it with the preceding block of ciphertext prior to encryption. This additional step increases both complexity and processing time, making AES-CBC less efficient in terms of encryption time.

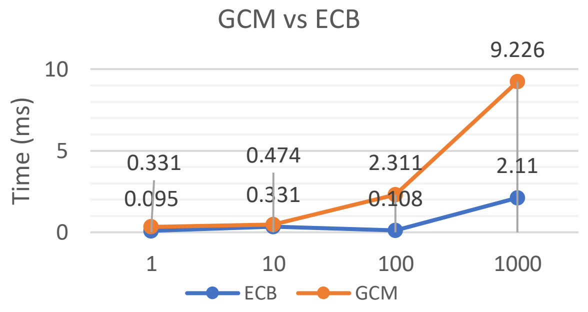

Figure 3. Comparison between fastest and second fastest AES modes.

Shown in figure 3 is comparison between fastest and second fastest AES modes.

AES-GCM exhibits a predictable increase in encryption time as file size grows, indicating that its performance is largely linear with respect to the amount of data processed. For very small inputs—for instance, 1 KB files—the observed time (approximately 0.331 ms) is influenced significantly by fixed overheads such as cipher initialization, generation of nonces, and setup for authenticated encryption. As the file size increases to 10 KB and beyond, these constant costs become less dominant, and the time required to process each additional block of data starts to reflect a near-linear increase. In practice, while the underlying authenticator (involving Galois field multiplication) introduces additional computational complexity compared to non-authenticated modes, the resulting slope, or per-KB processing time, remains consistent. This linearity means that when scaling up from 100 KB to 1000 KB, the increase in encryption time (from roughly 2.311 ms to 9.226 ms) is primarily due to the additional data being processed rather than non-scalable overhead, confirming that AES-GCM's runtime grows in direct proportion to data size after accounting for the base cost.

AES-ECB on the other hand, stands out as the fastest mode, primarily because it operates on each data block independently without any added overhead associated with chaining or authentication. The very low encryption time for a 1 KB file (approximately 0.095 ms) suggests that there is a small fixed overhead due to initialization and function call latency that does not scale much with file size for extremely small datasets. However, as file size increases, the encryption time for ECB mode follows a linear trend, with the processing time increment reflecting the addition of new blocks to the encryption pipeline. For example, moving from 10 KB to 100 KB and then to 1000 KB sees predictable increases, with timings of about 0.108 ms for 10 KB, 1.087 ms for 100 KB, and 2.77 ms for 1000 KB. This consistent addition per unit of data confirms that ECB's computational effort scales linearly with file size. Despite the attractive performance characteristics, it is important to note that the simplicity of ECB poses significant security risks such as revealing patterns in plaintext which fundamentally undermines its suitability for secure communications, despite its superior speed.

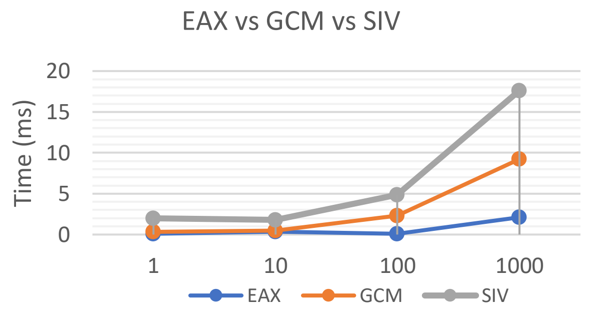

Figure 4. Comparison between the median AES modes.

Examining the performance of the middle ground result data in figure 4 reveals that AES-GCM, which falls between the fastest (AES-EAX) and the slowest (AES-SIV) modes, displays a near-linear increase in encryption time as file size grows. While AES-GCM benefits from authenticated encryption and efficient processing of larger data blocks through its counter mode and Galois field multiplication, the time difference compared to AES-EAX is marginal. For example, on 1 KB files, AES-GCM averages around 0.331 ms while AES-EAX clocks in at approximately 0.301 ms; and for 1000 KB files, AES-GCM records about 9.226 ms compared to AES-EAX's 10.706 ms. This similarity in performance indicates that AES-GCM does not yield a significant speed advantage over the already fast AES-EAX mode. For IoT devices where every microsecond counts in extending battery life and ensuring real-time responsiveness the extra overhead introduced by the authenticated encryption in AES-GCM may not deliver a clear performance benefit over alternatives that are marginally faster.

In resource-restricted environments where computational efficiency is paramount, the negligible performance difference between AES-GCM and AES-EAX becomes a deciding factor. For instance, in scenarios such as industrial IoT sensor networks or remote environmental monitoring, the choice of encryption should ideally maximize throughput while consuming minimal energy. Although AES-GCM offers robust security features, its median performance does not stand out when compared to modes like AES-EAX. In these cases, designers might opt for a mode that either delivers faster operations if the data sensitivity level allows for a slight relaxation in security measures or choose a mode that integrates security measures only when absolutely necessary. Overall, the lack of a clear advantage in throughput for AES-GCM suggests that its complexity does not translate into significant operational benefits on ultra-small and energy-constrained devices, making it less suitable in situations where every operation per second matters.

IV. CONCLUSION

For IoT devices operating under stringent resource constraints—such as limited processing power, memory, and battery life—selecting an efficient encryption mode is crucial. AES-GCM is highly recommended in these scenarios because it provides authenticated encryption, ensuring both the confidentiality and integrity of transmitted data. For example, smart sensors deployed in remote agricultural fields often use LoRa communication to transmit environmental data such as soil moisture and temperature. In these cases, AES-GCM's balanced approach minimizes computational overhead while delivering the security assurances required to prevent data tampering or spoofing, The performance of AES-GCM scales well with data size, making it ideal for applications that must process transmissions reliably under energy-constrained conditions. Additionally, Muttaqin et al. [20] discuss AES implementations on IoT modules, reinforcing that modes such as AES-GCM, which provide both confidentiality and data integrity while maintaining scalability with data size, are ideal for such applications. On the other hand, there are situations where security may be a secondary consideration compared to processing speed, particularly if the data itself is non-sensitive. AES-ECB offers remarkably fast encryption speeds because it processes each block of data independently without the overhead of chaining or authentication. In a practical scenario, a network of industrial IoT devices might periodically log ambient operational data where maintaining a high throughput is the primary concern. If the data is used solely for monitoring non-critical parameters such as ambient temperature or humidity, where occasional exposure of patterns is acceptable AES-ECB can be considered. However, one must remain cautious, as the absence of chaining can expose repeated patterns in the plaintext, leading to potential security vulnerabilities. In instances where the transmitted data does not require strong protection, and resource conservation is the overriding priority, the use of ECB mode can be justified. Habibi et al.[21] indirectly support the idea that simpler block processing (as in ECB) offers speed advantages. However, caution is advised because ECB's lack of chaining makes it less secure when encryption of patterned data is involved. Alternatively, AES-EAX stands as a favorable option when data integrity and authentication are paramount. Although slightly slower than AES-GCM, AES-EAX integrates both encryption and a message authentication code within a single process. This mode is particularly useful in applications like remote patient monitoring devices or critical industrial control systems, where even minimal data corruption might lead to severe consequences. These applications require that not only is the data kept confidential, but also that any unauthorized modification is immediately detectable. Therefore, while resource-constrained IoT deployments often benefit from the efficiency of AES-GCM, narrow cases where data manipulation pose a high risk may justify the additional overhead of AES-EAX. In conclusion, the choice of AES encryption mode in IoT devices should be tailored to the specific requirements of the application. For most resource-restricted environments, AES-GCM offers an optimal balance between speed and robust security, making it the preferred encryption mode. In contrast, AES-ECB may be acceptable for non-critical data where performance takes precedence, while AES-EAX is recommended for cases that demand a higher degree of data integrity and authentication. This nuanced approach ensures that IoT systems can remain energy-efficient and high-performing without compromising on essential security features.

REFERENCES

[1] R. M. Muchamad, A. Asriyanik, and A. Pambudi, "Implementasi Algoritma Advanced Encryption Standard (AES) untuk Mengenkripsi Datastore pada Aplikasi Berbasis Android," *Jurnal Mnemonic,* vol. 6, no. 1, pp. 55–64, 2023. doi: 10.36040/mnemonic.v6i1.5889.

[2] N. Mouha, "Review of the Advanced Encryption Standard," *National Institute of Standards and Technology (NIST)*, 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8319.pdf. [Accessed: Jun. 2025].

[3] A. Prameshwari and N. P. Sastra, "Implementasi Algoritma Advanced Encryption Standard (AES) 128 Untuk Enkripsi dan Dekripsi File Dokumen," *Eksplora Informatika*, vol. 8, no. 1, p. 52, 2018. doi: 10.30864/eksplora.v8i1.139.

[4] U. K. Prodhan, A. H. M. Shahariar, I. Hussain, F. Rumi, J. Kabi, and K. Nazrul, "Performance analysis of parallel implementation of Advanced Encryption Standard (AES) over serial implementation," *International Journal of Information Science and Technology*, vol. 2, no. 6, pp. 1–16, 2012. doi: 10.5121/IJIST.2012.2601.

[5] S. Kumar, "A review on high-speed Advanced Encryption Standard (AES)," *Journal of Emerging Technologies and Innovative Research*, vol. 5, no. 8, pp. 1255–1260, 2018. [Online]. Available: https://www.jetir.org/view?paper=JETIRA006219. [Accessed: Jun. 2025].

[6] M. P. Manuel and K. Daimi, "Implementing cryptography in LoRa-based communication devices for unmanned ground vehicle applications," *SN Applied Sciences*, vol. 3, no. 4, 2021. doi: 10.1007/s42452-021-04377-y.

[7] T. Taufiqqurrachman and D. Elsandi, "Security analysis and encryption time comparison on cryptography Advanced Encryption Standard (AES)," *Jurnal Inovatif*, vol. 5, no. 1, p. 60, 2022. doi: 10.32832/inova-tif.v5i1.8345.

[8] K. Muttaqin and J. Rahmadoni, "Analysis and design of file security system using AES cryptography," *Journal of Advanced Encryption and Technology Studies*, vol. 1, no. 2, pp. 113–123, 2020. doi: 10.37385/JAETS.V1I2.78.

[9] U. Pujeri, S. S. Desai, and A. Savyanavar, *Encryption Techniques for the Modern World*, IGI Global, 2020, pp. 285–319. doi: 10.4018/978-1-5225-8458-2.CH013.

[10] O. Agbelusi and M. Olumuyiwa, "Comparative analysis of encryption algorithms," *European Journal of Technology*, vol. 7, no. 1, pp. 1–9, 2023. doi: 10.47672/ejt.1312.

[11] N. Agnihotri and A. K. Sharma, "Comparative analysis of different symmetric encryption techniques based on computation time," *Grid Computing*, 2020. doi: 10.1109/PDGC50313.2020.9315848.

[12] B. Olivia et al., "Implementasi Kriptografi pada Keamanan Data menggunakan algoritma Advanced Encryption Standard (AES)," *Jurnal Simantec*, vol. 11, no. 2, pp. 167–174, 2023.

[13] A. Prameshwari and N. P. Sastra, "Implementasi Algoritma Advanced Encryption Standard (AES) 128 Untuk Enkripsi dan Dekripsi File Dokumen," *Eksplora Informatika*, vol. 8, no. 1, p. 52, 2018. doi: 10.30864/eksplora.v8i1.139.

[14] A. U. Rahman, S. U. Miah, and S. Azad, "Advanced encryption standard," *Practical Cryptography: Algorithms and Implementations Using C++*, pp. 91–126, 2014. doi: 10.1201/b17707.

[15] R. Ravida and H. A. Santoso, "Advanced Encryption Standard (AES) 128-bit untuk keamanan data Internet of Things (IoT) tanaman hidroponik," *Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi)*, vol. 4, no. 6, 2020. doi: 10.29207/resti.v4i6.2478.

[16] B. P. and A. Farisi, "Perbandingan kinerja algoritma kandidat AES dalam enkripsi dan dekripsi file dokumen," *MDP Student Conference*, vol. 2, no. 1, pp. 282–289, 2023. doi: 10.35957/mdp-sc.v2i1.4367.

[17] J. S. Schwarz, C. Chapman, and E. McDonnell Feit, "An overview of Python," in *Python for Marketing Research and Analytics*, Springer International Publishing, 2020, pp. 9–45. doi: 10.1007/978-3-030-49720-0_2.

[18] A. Bogdanov, "AES-based authenticated encryption modes in parallel high-performance software," *IACR*

Cryptology ePrint Archive, 2014. [Online]. Available: http://eprint.iacr.org/2014/186.pdf. [Accessed: Jun. 2025].

[19] C. Christopher, A. Gunawan, and S. Prima, “Encrypted short message service design using combination of modified AES and Vigenere cipher algorithm,” *EMACS Journal*, vol. 4, no. 2, pp. 73–77, 2022. doi: 10.21512/emacsjournal.v4i2.8273.

[20] H. D. Kotha, “AES encryption and decryption standards,” *Journal of Physics: Conference Series*, vol. 1228, no. 1, p. 012006, 2019. doi: 10.1088/1742-6596/1228/1/012006.

[21] National Institute of Standards and Technology, “Recommendation for block cipher modes of operation,” *NIST Special Publication 800-38A*, 2001. [Online]. Available: http://csrc.nist.gov/publications/drafts/800-38g/sp800_38g_draft.pdf. [Accessed: Jun. 2025].