Received December 23, 2025; accepted January 23, 2026. Date of publication June 30, 2026
Digital Object Identifier: https://doi.org/10/25047/jtit.v13i1.473

Comparative Analysis of AES-GCM and ChaCha20-Poly1305 in IoT Data Encryption Based on ESP32

DWI DINDA MEYLANI ANGELINA[1], RONALD DAVID MARCUS[2]

[1, 2]Universitas Merdeka Malang, Jl. Terusan Raya Dieng No. 62 – 64, Kota Malang, Jawa Timur 65146, Indonesia

CORESPONDING AUTHOR: RONALD DAVID MARCUS (email: ronald.mangero@unmer.ac.id)

ABSTRACT Internet of Things (IoT) devices have become an integral component in modern applications, but their resource constraints pose significant challenges in implementing robust security measures. Although AES-GCM and ChaCha20-Poly1305 are widely recognized as Authenticated Encryption with Associated Data (AEAD) algorithms, their comparative performance on resource-constrained microcontrollers such as the ESP32 remains under-explored, particularly in terms of execution time, memory usage, power consumption, and throughput. This research aims to conduct a comprehensive performance analysis of both algorithms in securing IoT data transmission on the ESP32-WROOM-32D microcontroller. This study implements both algorithms using Arduino IDE 2.3.6, utilizing ESP32 hardware acceleration for AES-GCM and an optimized software implementation for ChaCha20-Poly1305. Performance evaluation includes measuring execution time precision using the `micros()` function, memory usage through `ESP.getFreeHeap()`, and power consumption through shunt resistor analysis. The research results reveal different performance characteristics between the two algorithms across all evaluated metrics, providing valuable comparative insights for IoT developers to choose the optimal cryptographic solution based on specific application requirements and resource constraints, thereby enhancing security implementation in embedded systems.

KEYWORDS: AES-GCM, ChaCha20-Poly1305, ESP32

I. INTRODUCTION

The Internet of Things (IoT) has become an essential component in modern life, with rapid growth across various sectors such as smart homes [1][2]. The ESP32 is one of the most popular microcontrollers used in IoT device development due to its built-in Wi-Fi and Bluetooth capabilities, affordable price, and adequate performance [3][4]. Data transmitted by IoT devices is often sensitive and private in nature. Therefore, the use of appropriate encryption algorithms becomes crucial to maintain data confidentiality, integrity, and authentication [5][6]. AES-GCM and ChaCha20-Poly1305 are Authenticated Encryption with Associated Data (AEAD) algorithms that are widely recommended for secure communication on resource-constrained devices. Although both algorithms have been extensively studied in general, a comparison of their implementation on ESP32 microcontrollers in the context of IoT applications still requires more in-depth examination. Based on this background, this research aims to conduct a comprehensive comparative analysis between AES-GCM and ChaCha20-Poly1305 algorithms on the ESP32 microcontroller, in order to provide recommendations for the most optimal algorithm for data security on IoT devices with limited resource characteristics [7][8][9].

This research specifically focuses on AES-GCM and ChaCha20-Poly1305 for several critical reasons. First, both algorithms represent the two most widely adopted AEAD standards in modern security protocols, with AES-GCM standardized by NIST (SP 800-38D) and extensively used in TLS/SSL, while ChaCha20-Poly1305 is adopted in TLS 1.3 and widely implemented in mobile and embedded systems. Second, these algorithms embody fundamentally different cryptographic design philosophies: AES-GCM is block cipher-based and benefits from hardware acceleration on

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

many processors, whereas ChaCha20-Poly1305 is stream cipher-based and designed for efficient software implementation without requiring hardware acceleration [10][11][12]. This distinction is particularly relevant for ESP32, which features built-in AES hardware acceleration but lacks dedicated ChaCha20 support, making the performance comparison scientifically valuable. Third, both algorithms provide mature and well-optimized library implementations for ESP32, ensuring practical applicability of our findings. While other AEAD algorithms exist AES-GCM and ChaCha20-Poly1305 represent the primary choice faced by IoT developers and have the most extensive industry adoption, making this comparative analysis highly relevant for real-world IoT implementations.

Several studies have investigated the performance and security of AES-GCM and ChaCha20-Poly1305 algorithms in various contexts and platforms. Research conducted by Susanti et al. [13] entitled "Comparison of Performance and Security of Modern Cryptographic Algorithms AES-GCM and ChaCha20-Poly1305" analyzed the comparison between the performance and security levels of both algorithms through testing of encryption and decryption time, as well as simulations of timing attacks to evaluate the effectiveness of each algorithm in maintaining data security. Nguyen and Le [14] in their study "Implementation of ChaCha20-Poly1305 on Self-Organization Data Framing for Enhancing IoT Communication" implemented the ChaCha20-Poly1305 algorithm in a data framing mechanism to enhance communication security in IoT systems, with evaluation performed on ARM Cortex-M4 (STM32) and ESP32-based devices to assess the performance and reliability of the algorithm in IoT environments. Furthermore, Patria et al. [15] conducted a "Comparative Analysis of the Performance of AES-GCM and ChaCha20-Poly1305 in AEAD-Based PDF Document Encryption" which compared the performance of both algorithms in encrypting PDF documents using the AEAD scheme, with evaluation carried out on servers equipped with AES-NI-enabled CPUs to identify differences in efficiency and performance between the two algorithms.

TABLE 1. Previous Research

Researcher's Name/Year	Research Title	Research Result	Equation	Difference
Anggi Susanti, Bayu Ade Prasetiya, Oktafiyan Dyah Pangesti, Liling Daru Suryawati, Indrawan Ady Saputro, 2024	Perbandingan Kinerja Dan Keamanan Algoritma Kriptografi Modern AES-GCM Dengan ChaCha20-Poly1305	This study analyzes the comparison between the performance and security levels of AES-GCM and ChaCha20-Poly1305 through encryption/decryption time testing and timing attack simulations.	This study tests both the AES-GCM and ChaCha20-Poly1305 algorithms using quantitative experimental methods to evaluate encryption and decryption performance as well as security against timing attacks.	This study used a standard desktop/PC platform with Python programming language and data sizes of 1KB, 1MB, and 10MB. The results show that ChaCha20 is faster with a focus on general comparisons.
Vu-Minh-Thanh Nguyen, Duc-Hung Le, 2024	Implementation of ChaCha20-Poly1305 on Self-Organization Data Framing for Enhancing IoT Communication	This research implements ChaCha20-Poly1305 on framing data for IoT communication. Tested on ARM Cortex-M4 (STM32) and ESP32.	This study also tested ChaCha20-Poly1305 with a focus on data security and performance and efficiency evaluation using the AEAD scheme.	This research uses an IoT embedded systems platform (ARM Cortex-M4) with C/C++ language and focuses on IoT communication with UART/MQTT. The overhead file is 28 bytes for a specific BTS monitoring application.

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Muhammad Patria, Dea Andini Andriati, 2025	Analisis Komparatif Performa AES-GCM dan ChaCha20-Poly1305 dalam Enkripsi Dokumen PDF Berbasis AEAD	This study compares the performance of AES-GCM and ChaCha20-Poly1305 in encrypting PDF documents. Tested on a server with an AES-NI CPU.	This study compares AES-GCM vs ChaCha20-Poly1305 using experimental methods to test encryption and decryption times and evaluate throughput and stability.	This study compares AES-GCM vs ChaCha20-Poly1305 using experimental methods to test encryption and decryption times as well as evaluate throughput and stability. The platform used is a modern VPS Server with AES-NI using the Go (Golang) language with data sizes ranging from 100KB to 200MB (5000+ files). The results show that AES-GCM is faster with a focus on encrypting PDF documents for the e-Meterai platform application.
Ahmad Miftah Fauzi, Arief Zulianto, Purnomo Yustianto, 2025	Perbandingan Kinerja Algoritma Enkripsi Chacha20, Salsa20, dan Advanced Encryption Standard (AES) pada Mikrokontroler	This study compares AES, ChaCha20, and Salsa20 on four microcontroller platforms (ESP32, ESP32-S2, Arduino Uno, Arduino Mega) using Wokwi simulation with 30 repetitions.	This study tests both the ChaCha20 and AES algorithms, focusing on cryptographic performance using experimental methods to evaluate speed and efficiency.	This study uses a microcontroller platform (ESP32-S2, Arduino) to test three algorithms (AES, ChaCha20, Salsa20) with a focus on embedded systems with limited power and smaller data sizes. The results show that ChaCha20 is the most efficient.

II. METHOD

A. RESEARCH DESIGN

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

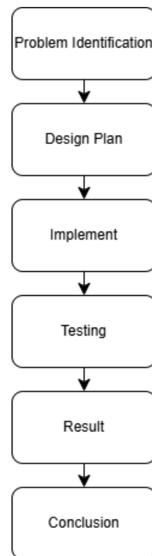

FIGURE 1. Research Design

This research, as illustrated in Figure 1, began with the identification of the problem regarding the need to select an efficient and secure cryptographic algorithm for IoT devices with limited resources such as the ESP32 microcontroller, followed by the design of a framework that systematically and methodically addresses this problem. During the implementation phase, AES-GCM utilized the hardware acceleration available on the ESP32, while ChaCha20-Poly1305 was optimally implemented in software, with a measurement system designed to capture real-time data on execution time, memory usage, and power consumption with high precision. Next, the testing stage was carried out by systematically executing pre-prepared scenarios to obtain performance data for both algorithms, where the results of the analysis were used as a basis for comparing the performance of the two algorithms in various parameters that had been tested. The research conclusions include recommendations for practical implementation for IoT application developers, identification of research limitations, and suggestions for further development and research in similar areas.

B. DESIGN OF THE DEVICE

The IoT device design in this study, shown in Figure 2, was designed to simulate a simple IoT application scenario involving data encryption on devices with limited resources. The IoT system consists of an ESP32 module as the processing core, which is connected to supporting components for performance measurement and status indication.

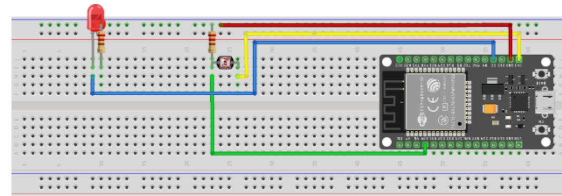

FIGURE 2. Design of The Device

C. SYSTEM BLOCK DIAGRAM

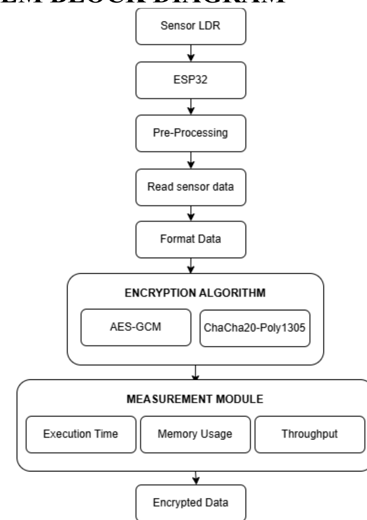

FIGURE 3. System Block Diagram

This system block diagram, shown in Figure 3, illustrates the workflow from sensor data acquisition to encryption on ESP32-based IoT devices. The LDR sensor detects light intensity and generates analog data that is received by the ESP32 microcontroller, then the data goes through a pre-processing stage for filtering and analog-to-digital conversion. After the sensor data is read and formatted into the appropriate structure, it enters the Encryption Algorithm module, which applies two methods: AES-GCM with hardware acceleration and ChaCha20-Poly1305 in software. During the encryption process, the Measurement Module monitors three performance parameters, namely Execution Time, Memory Usage, and Throughput, to compare the efficiency of the two algorithms. The end result of this process is Encrypted Data that is secure for transmission or storage, along with performance measurement data that is used as comparative analysis material for the two encryption algorithms.

D. DATA COLLECTION TECHNIQUES

Data collection was conducted through direct experiments using the ESP32 platform with integrated instrumentation. Execution time was measured using the micros() function in the Arduino IDE with microsecond precision, recording the time before and after the encryption/decryption process

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

for payloads of 16, 32, and 64 bytes representing sensor data. RAM memory measurements were performed using the ESP.getHeapSize() and ESP.getFreeHeap() functions to monitor the heap, as well as an IDE profiling tool for static memory analysis, which was measured at three stages: during booting, algorithm initialization, and during the encryption/decryption process. Power consumption was measured by installing a 1Ω shunt resistor in series on the ESP32 power supply line, using a digital multimeter to calculate the current in idle and intensive encryption conditions, with the ESP32 operating in full active mode without Wi-Fi and Bluetooth, and recording the current every 100 milliseconds for 30 seconds. Throughput is calculated from the comparison of plaintext data size with execution duration in bytes per second (B/s) for various data size variations to analyze the algorithm's scalability in simulating various data transmission scenarios in IoT systems.

III. RESULT AND DISCUSSION

A. HARDWARE IMPLEMENTATION

The hardware is assembled based on the specified system block diagram. The ESP32 DevKit, specifically the ESP32-WROOM-32D type, serves as the core of cryptographic processing. As an external trigger, an LDR sensor is installed to ensure consistency in the initiation of each measurement cycle. Meanwhile, an indicator LED connected to the GPIO pin provides a visual display of the system's operating status.

The Figure 4 shows the hardware design implemented during the experimental testing phase. The ESP32 DevKit module is placed on the right side of the breadboard, with power and serial communication supplied via a micro USB cable. This supports controlled testing that is consistently replicable.

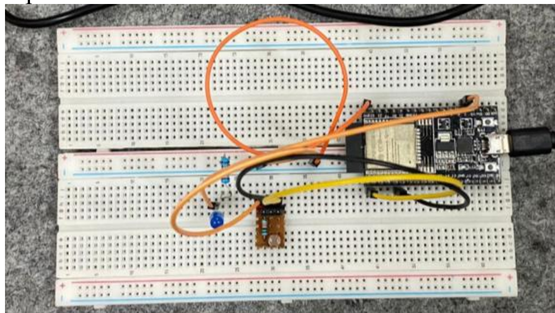

FIGURE 4. Hardware Implementation

B. SOFTWARE IMPLEMENTATION

The software was developed using Arduino IDE 2.3.6 with ESP32 Board Package 3.3.5 support. Its main functions are to control sensor data acquisition, data processing, and wireless communication. Sensor readings are taken periodically using the `analogRead()` function, and the raw data obtained is then processed according to system requirements. Before being sent to the server or displayed, the data is secured using cryptographic techniques to maintain confidentiality and integrity. The system also records performance parameters such as execution time, memory usage, throughput, and estimated power consumption.

The results of data processing are presented in a structured format in the Serial Monitor, facilitating recording and further analysis with software such as Microsoft Excel. With the implemented architecture, the system is expected to operate stably, efficiently, and in accordance with the planned functional specifications.

1. **Implementation of the AES-GCM Algorithm**

a) Library and Dependencies

1) Library: mbedTLS (Mbed Transport Layer Security)

2) Header files:
   - mbedtls/gcm.h - for Galois/Counter Mode
   - mbedtls/aes.h - for core AES algorithm

3) Supoorting libraries:
   - WiFi.h - for ESP32 wireless connectivity

b) Program Architecture

1) Initialization Module

During the setup stage, the following components are initialized:

```
244 // ===== SETUP =====
245 void setup() {
246   Serial.begin(115200);
247   delay(1000);
248
249   pinMode(LDR_PIN, INPUT);
250   pinMode(LED_PIN, OUTPUT);
251   analogReadResolution(12);
```

FIGURE 5. Setup Module

Cryptographic initialization is performed with the following steps:

```
256   // ====== INISIALISASI AES-GCM ======
257   mbedtls_gcm_init(&aes_ctx);
258   mbedtls_gcm_setkey(&aes_ctx, MBEDTLS_CIPHER_ID_AES, aes_key, 256);
```

FIGURE 6. AES-GCM Initialization

Encryption Key Configuration:

- Key length: 256-bit (32 bytes) for maximum security level
- Key value: Uses sequential values from 0x00 to 0x1F for consistent testing
- IV (Initialization Vector): 12 bytes with automatic increment mechanism to prevent reuse of the same IV

2) Data Reading and Encryption Module

This module is the core of the performance measurement system. The updateSensorData() function performs a series of operations with high-precision time measurement:

- Sensor Data Acquisition

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

```
// Baca sensor
unsigned long startSensor = micros();
nilaiLDR = analogRead(LDR_PIN);
ledStatus = (nilaiLDR < threshold);
digitalWrite(LED_PIN, ledStatus ? HIGH : LOW);
unsigned long timeSensor = micros() - startSensor;
```

FIGURE 7. Sensor Data Acquisition

- Plaintext Formation

The system creates plaintext with a structured format that includes sensor information:

```
// ====== BUAT PLAINTEXT 32 BYTES ======
char plaintext[64];
sprintf(plaintext, "LDR:%04d|LED:%d|TIME:%08lu", nilaiLDR, ledStatus, millis());
lastPlaintext = String(plaintext);
```

FIGURE 8. Plaintext Formation

To meet the test data size requirement (32 bytes), deterministic padding implementation is performed:

```
// PADDING KE 32 BYTES
size_t len = strlen(plaintext);
int targetSize = 32;  // Target: 32 bytes

while(len < targetSize) {
  plaintext[len] = 'X';  // Tambah padding 'X'
  len++;
}
plaintext[targetSize] = '\0';  // Tutup string
len = targetSize;  // Set panjang = 32 bytes

lastPlaintext = String(plaintext);  // Update untuk display
```

FIGURE 9. Plaintext Padding

- AES-GCM Encryption

The encryption process is performed with precise execution time measurement:

```
unsigned char ciphertext[32];  // 32 bytes untuk ciphertext
unsigned char tag[16];

unsigned long startEncrypt = micros();

// Enkripsi dengan AES-GCM
int ret = mbedtls_gcm_crypt_and_tag(
  &aes_ctx,
  MBEDTLS_GCM_ENCRYPT,
  len,              // 32 bytes
  iv, 12,           // IV 12 bytes
  NULL, 0,          // No additional authenticated data
  (const unsigned char*)plaintext,
  ciphertext,
  16,               // Tag length
  tag
);

unsigned long timeEncrypt = micros() - startEncrypt;
```

FIGURE 10. AES-GCM Encryption

Parameter Explanation:

- IV (12 bytes): Uses increment mechanism for each encryption, ensuring unique IV for each operation
- AAD (Additional Authenticated Data): Not used in this research (NULL, 0), focus on pure sensor data encryption
- Tag (16 bytes): Generates 128-bit authentication tag for integrity verification

2. Implementation of the ChaCha20-Poly1305 Algorithm

a) Library and Dependencies

1) Library: mbedTLS (Mbed Transport Layer Security)
2) Header files:
   - WiFi.h - for ESP32 wireless connectivity
   - WebServer.h - for web-based monitoring interface
3) Supoorting libraries:
   - WiFi.h - for ESP32 wireless connectivity

b) Program Architecture

1) Initialization Module

During the setup stage, the following components are initialized:

```
// ================ SETUP ================
void setup() {
  Serial.begin(115200);
  delay(1000);

  pinMode(LDR_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT);
  analogReadResolution(12);
```

FIGURE 11. Setup Module

Unlike some other cryptographic implementations that require special context initialization in the setup stage, ChaCha20-Poly1305 uses a more flexible approach:

```
// AEAD object
ChaChaPoly aead;
```

FIGURE 12. Context Initialization

Encryption Key Configuration:

```
// 256-bit key
uint8_t key[32] = {
  0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
  0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,
  0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,
  0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F
};
```

FIGURE 13. Encryption Key Configuration

Key Specifications:

- Key length: 256-bit (32 bytes) for maximum security level
- Key value: Uses sequential values from 0x00 to 0x1F for consistent testing
- Key management: Key is reset for each encryption operation using setKey() function

Nonce (Initialization Vector) Configuration:

```
// 96-bit nonce
uint8_t nonce[12] = {0};
```

FIGURE 14. Nonce Configuration

Nonce Specifications:

- Size: 12 bytes (96 bits)
- Mechanism: Counter-based increment for each encryption
- Security: Ensures nonce uniqueness, critical for ChaCha20-Poly1305 security

2) Data Reading and Encryption Module

- Sensor Data Acquisition

```
// Increment nonce (untuk keamanan)
nonce[11]++;
```

FIGURE 15. Sensor Data Acquisition

- Plaintext Formation and Padding

```
// Baca sensor
unsigned long startSensor = micros();
nilaiLDR = analogRead(LDR_PIN);
ledStatus = (nilaiLDR < threshold);
digitalWrite(LED_PIN, ledStatus ? HIGH : LOW);
unsigned long timeSensor = micros() - startSensor;
```

FIGURE 16. Plaintext Formation and Padding

- ChaCha20-Poly1305 Encryption

The encryption process consists of 5 main stages with precision time measurement:

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

a. Context Initialization and Key Setup

```
// ====== BUAT PLAINTEXT 64 BYTES ======
char plaintext[128];
sprintf(plaintext, "LDR:%04d|LED:%d|TIME:%08lu|SENSOR:ACTIVE|STATUS:OK", nilaiLDR,
lastPlaintext = String(plaintext);
```

FIGURE 17. Context Initialization

b. Nonce Increment and Configuration

```
// PADDING KE 64 BYTES
size_t len = strlen(plaintext);
int targetSize = 64;  // Target: 64 bytes

while(len < targetSize) {
  plaintext[len] = 'X';  // Tambah padding 'X'
  len++;
}
plaintext[targetSize] = '\0';  // Tutup string
len = targetSize;  // Set panjang = 64 bytes
```

FIGURE 18. Nonce Configuration

c. Data Encryption

```
aead.clear();
aead.setKey(key, sizeof(key));
aead.setIV(nonce, sizeof(nonce));
aead.encrypt(ciphertext, (uint8_t*)plaintext, len);
aead.computeTag(tag, sizeof(tag));

unsigned long timeEncrypt = micros() - startEncrypt;
```

FIGURE 19. Data Encryption

C. TEST RESULTS

TABLE 2. Execution Time Test Results

Algorithm	Minimum Value	Maximum Value	Average Value
AES-GCM	175 ms	226 ms	176,65 ms
ChaCha20-Poly1305	112 ms	131 ms	119,24 ms

ChaCha20-Poly1305 is 57.41 ms faster than AES-GCM, or approximately 32.5% faster in execution time. This significant performance improvement can be attributed to ChaCha20-Poly1305's software-optimized design, which does not require specialized hardware acceleration like AES-NI instructions that benefit AES-GCM. The faster execution time makes ChaCha20-Poly1305 particularly suitable for real-time encryption applications and environments with limited hardware resources.

TABLE 3. Memory Usage Test Results

Algorithm	Minimum Value	Maximum Value	Average Value
AES-GCM	96.864 KB	96.888 KB	96.868,67 KB
ChaCha20-Poly1305	96.788 KB	96.992 KB	96.862,12 KB

ChaCha20-Poly1305 uses 6.55 KB less than AES-GCM, or 0.0068% less memory. However, this difference is negligible in practical applications and indicates that both algorithms have nearly identical memory footprints. This minimal variance suggests that memory consumption should not be a determining factor when choosing between these two algorithms, as both demonstrate excellent memory efficiency suitable for resource-constrained devices.

TABLE 4. Power Consumption Test Results

Algorithm	Minimum Value	Maximum Value	Average Value
AES-GCM	80 mA	80 mA	80 mA
ChaCha20-Poly1305	80 mA	80 mA	80 mA

There is no difference in power consumption between the two algorithms, with both maintaining a consistent 80 milliampere. This identical energy consumption indicates that neither algorithm offers an advantage in terms of power efficiency for this implementation. The result suggests that the choice between AES-GCM and ChaCha20-Poly1305 will not impact the energy costs or battery life in mobile or IoT applications.

TABLE 5. Throughput Test Results

Algorithm	Minimum Value	Maximum Value	Average Value
AES-GCM	76.086,95 Kbps	111.111,11 Kbps	106.567,82 Kbps
ChaCha20-Poly1305	215.686,28 Kbps	269.230,78 Kbps	260.678,94 Kbps

ChaCha20-Poly1305 achieves a throughput of 260,678.94 Kbps, which is 154,111.12 Kbps higher than AES-GCM (106,567.82 Kbps), representing a 144.6% increase or approximately 2.4 times faster. This substantial throughput advantage demonstrates ChaCha20-Poly1305's superior capability in handling large volumes of data efficiently. The higher throughput makes ChaCha20-Poly1305 the preferred choice for applications requiring high-speed data transmission, such as video streaming, large file transfers, or high-traffic network communications.

The significantly higher throughput of ChaCha20-Poly1305 compared to AES-GCM can be attributed to several fundamental architectural differences between the two algorithms. First, ChaCha20-Poly1305 is a stream cipher that operates on individual bytes and uses simple ARX (Addition-Rotation-XOR) operations, which are inherently fast on general-purpose processors and require fewer CPU cycles per byte processed. In contrast, AES-GCM is a block cipher operating on fixed 128-bit blocks through multiple rounds of substitution-permutation network transformations, which involve table lookups and more complex operations that can create processing bottlenecks, especially when hardware acceleration is not available or fully utilized. Second, on the ESP32 platform used in this study, while AES-GCM can leverage hardware acceleration through the ESP32's built-in AES

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

cryptographic accelerator, this acceleration primarily benefits the AES encryption component but does not extend to the GHASH authentication function in GCM mode, which must still be computed in software using finite field arithmetic that is computationally intensive. ChaCha20-Poly1305, despite running entirely in software without dedicated hardware support, benefits from its design optimization for software implementation where the ChaCha20 cipher and Poly1305 authenticator both use simple 32-bit operations that execute efficiently on the ESP32's 32-bit architecture. Third, the memory access patterns differ significantly: AES-GCM's table-based S-box operations can suffer from cache misses and timing variations, whereas ChaCha20-Poly1305's operations are more cache-friendly with predictable, constant-time execution paths that reduce memory latency. Additionally, the parallel processing capability of ChaCha20-Poly1305 allows for better utilization of modern CPU architectures, as its quarter-round function can process multiple data streams simultaneously without dependencies, while AES-GCM's sequential block processing creates inherent bottlenecks. These combined factors explain why ChaCha20-Poly1305 demonstrates superior throughput performance even when AES-GCM has access to hardware acceleration, particularly in scenarios involving continuous data streaming where the cipher's ability to process data with minimal overhead becomes critical for maintaining high throughput rates.

TABLE 6. Comparative Summary Table

Parameter	AES-GCM	ChaCha20-Poly1305
Execution Time (ms)	176.65	119.24
Memory Usage (KB)	96868.67	96862.12
Power Consumption (W)	80	80
Throughput (Kbps)	106,567.82	260,678.94

Based on the comparison results in the table above, ChaCha20-Poly1305 shows superior performance in three of the four parameters tested. This algorithm is significantly faster in execution time (32.5% improvement) and delivers substantially higher throughput (2.4x faster) than AES-GCM. Meanwhile, the memory usage difference is minimal and power consumption remains identical at 80 milliampere.

The very small difference in memory usage indicates that both algorithms are equally efficient in terms of memory utilization. The major differentiating factor between the two algorithms lies in execution speed and throughput capacity, where ChaCha20-Poly1305 demonstrates clear advantages for high-volume data processing scenarios.

These findings align with previous research that have shown ChaCha20-Poly1305's efficiency in software-based implementations, particularly on devices without AES hardware acceleration. The performance gap becomes especially significant in mobile and embedded systems, where AES-GCM cannot leverage specialized CPU instructions. However, it should be noted that in systems with AES-NI support, the performance difference may be reduced, though ChaCha20-Poly1305 would still maintain its throughput advantage.

D. BRIEF ANALYSIS OF TEST RESULTS

1) Execution Time:
   ChaCha20-Poly1305 is significantly faster than AES-GCM, making it suitable for applications that require high speed such as streaming or real-time communication.
2) Memory Usage:
   The difference in memory usage is very small, but ChaCha20-Poly1305 is slightly more efficient.
3) Power Consumption:
   Both algorithms have the same power consumption, indicating equivalent energy efficiency.
4) Throughput:
   ChaCha20-Poly1305 shows significantly higher throughput performance, indicating its ability to handle large data volumes more efficiently.

IV. CONCLUSION

This research successfully analyzed the performance of AES-GCM and ChaCha20-Poly1305 on ESP32-WROOM-32D microcontroller to provide comparative insights for optimal cryptographic solution selection in resource-constrained IoT devices. The results demonstrate that ChaCha20-Poly1305 significantly outperforms AES-GCM with 32.5% faster execution time (119.24 ms vs. 176.65 ms) and 144.6% higher throughput (260,678.94 Kbps vs. 106,567.82 Kbps), while both algorithms show identical power consumption (80 milliampere) and negligible memory differences. These findings recommend ChaCha20-Poly1305 as the optimal choice for ESP32-based IoT applications requiring high-speed data processing and real-time encryption without specialized hardware acceleration.

REFERENCE

[1] I. Rozlomii, A. Yarmilko, and S. Naumenko, "Data security of IoT devices with limited resources: challenges and potential solutions," *CEUR Workshop Proc.*, vol. 3666, pp. 85–96, 2024, doi: 10.55056/jec.748.

[2] Adianto *et al.*, "Pengenalan Sistem IoT Pada Pemanfaatan Kebutuhan Sehari-Hari," *J. Cakrawala Marit.*, vol. 7, no. 1, pp. 1–12, 2024, doi: 10.35991/jcm.v7i1.13.

[3] M. Santo Gitakarma, K. Udy Ariawan, and I. G. M. S. Bumi Pracasitaram, "Peran mikrokontroler dalam pengembangan aplikasi IoT: tinjauan konseptual dan implementasi," *J. Komput. dan Teknol. Sains*, vol. 3, no. 2, pp. 18–24, 2024, doi: 10.37637/komteks.v3i2.2231.

[4] K. M. W. Hidayat and M. G. Al-Faris, "IoT-Based

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Electrical Power Consumption Monitoring System in Households Using ESP32 and PZEM-004T," *Brill. Res. Artif. Intell.*, vol. 5, no. 2, pp. 1077–1081, 2025, doi: 10.47709/brilliance.v5i2.6368.

[5] F. Izul Falad, E. L. Wibowo, and N. D. Damayanti, "Optimizing Security and Data Privacy in IoT Systems to Prevent Cyberattacks," *Innov. J. Soc. Sci. Res.*, vol. 3, pp. 10491–10496, 2023, doi: 10.31004/innovative.v3i4.16340.

[6] J. P. Aljabbar, M. R. F. Rajaguguk, and N. Wijaya, "Studi Literatur Keamanan Data Dalam Sistem Smart Home Berbasis Iot," *Device J. Inf. Syst. Comput. Sci. Inf. Technol.*, vol. 6, no. 1, pp. 245–255, 2025, doi: 10.46576/device.v6i1.6660.

[7] F. De Santis, A. Schauer, and G. Sigl, "ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications," *Proc. 2017 Des. Autom. Test Eur. DATE 2017*, pp. 692–697, 2017, doi: 10.23919/DATE.2017.7927078.

[8] V. R. Kebande, "Extended-Chacha20 Stream Cipher With Enhanced Quarter Round Function," *IEEE Access*, vol. 11, no. September, pp. 114220–114237, 2023, doi: 10.1109/ACCESS.2023.3324612.

[9] M. S. Mahdi, N. F. Hassan, and G. H. Abdul-Majeed, "An improved chacha algorithm for securing data on IoT devices," *SN Appl. Sci.*, vol. 3, no. 4, pp. 1–9, 2021, doi: 10.1007/s42452-021-04425-7.

[10] S. Gueron, "Vectorization of Poly1305 Message Authentication Code," pp. 145–150, 2015, doi: 10.1109/ITNG.2015.28.

[11] R. Ridho, C. Bisri, and A. M. Harahap, "Penerapan Kriptografi Enkripsi Dan Deskripsi Dalam Pendataan Pasien Klinik Mama Harfas Tembung Menggunakan Visual Basic," *J. Penelit. Dan …*, vol. 2, no. 1, pp. 30–33, 2023, doi: 10.47233/jppie.v2i1.676.

[12] N. Xie, Z. Gong, Y. Tang, and L. Wang, "Protecting White-Box Block Ciphers with Galois / Counter Mode," *IEEE Conf. Dependable Secur. Comput.*, 2022, doi: 10.1109/DSC54232.2022.9888845.

[13] A. Susanti, B. A. Prasetiya, O. D. Pangesti, L. D. Suryawati, and I. A. Saputro, "Perbandingan Kinerja dan Keamanan Algoritma Kriptografi Modern AES-GCM dengan CHACHA20-POLY1305," *Infomatek*, vol. 26, no. 2, pp. 253–264, 2024, doi: 10.23969/infomatek.v26i2.19255.

[14] V. Nguyen and D. Le, "Implementation of ChaCha20-Poly1305 on Self-Organization Data Framing for Enhancing IoT Communication," *REV J. Electron. Commun.*, vol. 14, no. 4, pp. 9–18, 2024, doi: 10.21553/rev-jec.368.

[15] M. Patria and D. A. Andriati, "Analisis Komparatif Performa AES-GCM dan ChaCha20-Poly1305 dalam Enkripsi Dokumen PDF Berbasis AEAD," *Arcitech*, vol. 5, no. 1, pp. 49–69, 2025, doi: 10.29240/arcitech.v5i1.13645.

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/