International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

A Python-Based Simulation and Security Analysis of ChaCha20 and AES for Secure Data Transmission in Resource-Constrained Environments

Jechonia M. Dungog

Master of Science in Computer Science, North Eastern Mindanao State University-Tandag City

Abstract

In response to the growing need for efficient cryptographic mechanisms within resource-constrained environments, this study presents a comparative analysis of two widely adopted encryption algorithms: Advanced Encryption Standard (AES) and ChaCha20. Using Python-based simulations, the research evaluates each algorithm's performance based on encryption time, decryption time, and peak memory consumption across varying file sizes. The experimental setup mimics real-world constraints typical of IoT devices and embedded systems. Results indicate that ChaCha20 consistently outperforms AES in terms of speed and memory efficiency, making it a more suitable option for low-resource environments. AES, while secure and robust, demonstrates increasing computational and memory overhead with larger input sizes. The findings support the adoption of lightweight cryptographic algorithms like ChaCha20 in latency-sensitive and memory-limited applications, offering valuable guidance for developers and system architects aiming to optimize security without compromising performance.

Keywords: ChaCha20, AES, lightweight cryptography, resource-constrained environments, Python simulation

1. Introduction

In today's digital landscape, the ubiquity of resource-constrained environments—particularly within the scope of the Internet of Things (IoT)—highlights the profound necessity for effective cryptographic solutions. With the increasingly pervasive nature of connected devices, the demand for lightweight cryptography, which balances performance and security in low-resource situations, has surged. Traditional cryptographic algorithms such as AES, while robust in security, are often inefficient in their resource consumption, rendering them less suitable for implementation in devices with limited processing power and memory [1][2]. This study focuses on the simulation and comparative analysis of ChaCha20 and AES encryption algorithms using Python, aiming to evaluate their performance and security in resource-constrained environments.

Previous literature has emphasized the significance of lightweight cryptographic solutions in the context of IoT and similar applications. For instance, ChaCha20 has been praised for its high-speed encryption and minimal resource consumption, making it an effective alternative to AES in constrained scenarios [3][4]. Research findings show that its design, as a stream cipher, allows for secure data transmission

without the computational burden of traditional block ciphers [3]. Moreover, the importance of integrating lightweight encryption standards into systems with limited processing capacity has been underlined by several studies that advocate for their widespread adoption in IoT infrastructures [5]. Additionally, initiatives by the National Institute of Standards and Technology (NIST) have set benchmarks and provided guidance for evaluating lightweight cryptographic algorithms, further validating their relevance in modern cryptographic research [2][6].

Despite the growing body of research, there remains a gap in comparative studies that simulate and assess both AES and ChaCha20 under identical conditions tailored to resource-limited environments. While each algorithm has been evaluated independently in prior works [4][7], limited efforts have been made to simulate their performance side-by-side using a standardized platform such as Python. Furthermore, there is a lack of in-depth analysis that not only benchmarks their computational efficiency but also assesses their viability in securing real-time data transmission on lightweight systems.

This study holds significant value in the current technological context, where the demand for secure and efficient encryption in constrained environments continues to rise. By simulating ChaCha20 and AES using Python and analyzing their respective performance metrics, the research provides critical insights into their applicability for secure data transmission. The findings are expected to guide developers, researchers, and IoT system designers in selecting suitable cryptographic standards that align with both security requirements and hardware limitations [8][9].

2. Methodology

This study employed a **quantitative experimental design** to evaluate and compare the performance of two cryptographic algorithms—**AES** and **ChaCha20**—in a simulated Python environment. The methodology involves algorithm implementation, controlled testing on varying file sizes, and graphical analysis of encryption and decryption performance metrics.

2.1. Algorithm Implementation

The cryptographic algorithms were implemented using the Python programming language, leveraging secure libraries such as:

1. cryptography – for AES and ChaCha20 primitives
2. pycryptodome – for low-level AES-CBC operations
3. time, psutil, and tracemalloc – for tracking execution time, CPU usage, and memory consumption

AES-256 was configured in CBC (Cipher Block Chaining) mode with PKCS7 padding, while ChaCha20 used a 256-bit key with a 96-bit nonce via the ChaCha20Poly1305 construction. The simulation script executed both encryption and decryption routines on the same data to ensure fairness in measurement.

2.2. Test Environment and Setup

All simulations were conducted on a single personal device to maintain consistency. The system specifications are:

1. Device: MSI GF63 Thin 11UC
2. Processor: Intel Core i5-11260H @ 2.60GHz (12 CPUs)
3. RAM: 16GB
4. Operating System: Windows 11 Home, 64-bit
5. Python Version: 3.x
6. Tools Used: Visual Studio Code, Windows Terminal

Five different input sizes—1KB, 100KB, 1MB, 5MB, and 10MB—were used to simulate a range of transmission conditions, from small IoT messages to larger file transfers. Random byte arrays (os.urandom) were generated as sample data.

2.3. Performance Metrics

The simulation script captured the following quantitative metrics for both AES and ChaCha20:

1. Encryption Time (seconds)
2. Decryption Time (seconds)
3. Peak Memory Usage (MB)

The time.time() module was used to record timing, while tracemalloc monitored memory allocation during execution.

2.4. Data Collection and Output

For each file size, the script logged results into a structured CSV file namedcrypto_benchmark_results.csv. The output included:

1. File size (KB)
2. Encryption/decryption time for AES and ChaCha20
3. Peak memory consumption

This structured data served as the basis for subsequent analysis and visualization.

2.5. Visualization and Analysis

To better interpret the comparative performance, a line chart was generated using the matplotlib library. The chart displayed:

1. Encryption and decryption time trends for AES and ChaCha20
2. X-axis: File sizes in KB
3. Y-axis: Time in seconds

This visual comparison helped highlight which algorithm performs more efficiently as file size increases. The analysis was used to identify practical trade-offs in performance, particularly for use in resource-constrained environments like IoT devices.

3. Results and Discussion

This chapter presents the experimental results derived from the simulation and benchmarking of the AES and ChaCha20 encryption algorithms. The study evaluates the encryption and decryption time, as well as peak memory usage, across various file sizes ranging from 1KB to 10MB. All tests were performed using Python on a controlled single-device setup to simulate constrained environments.

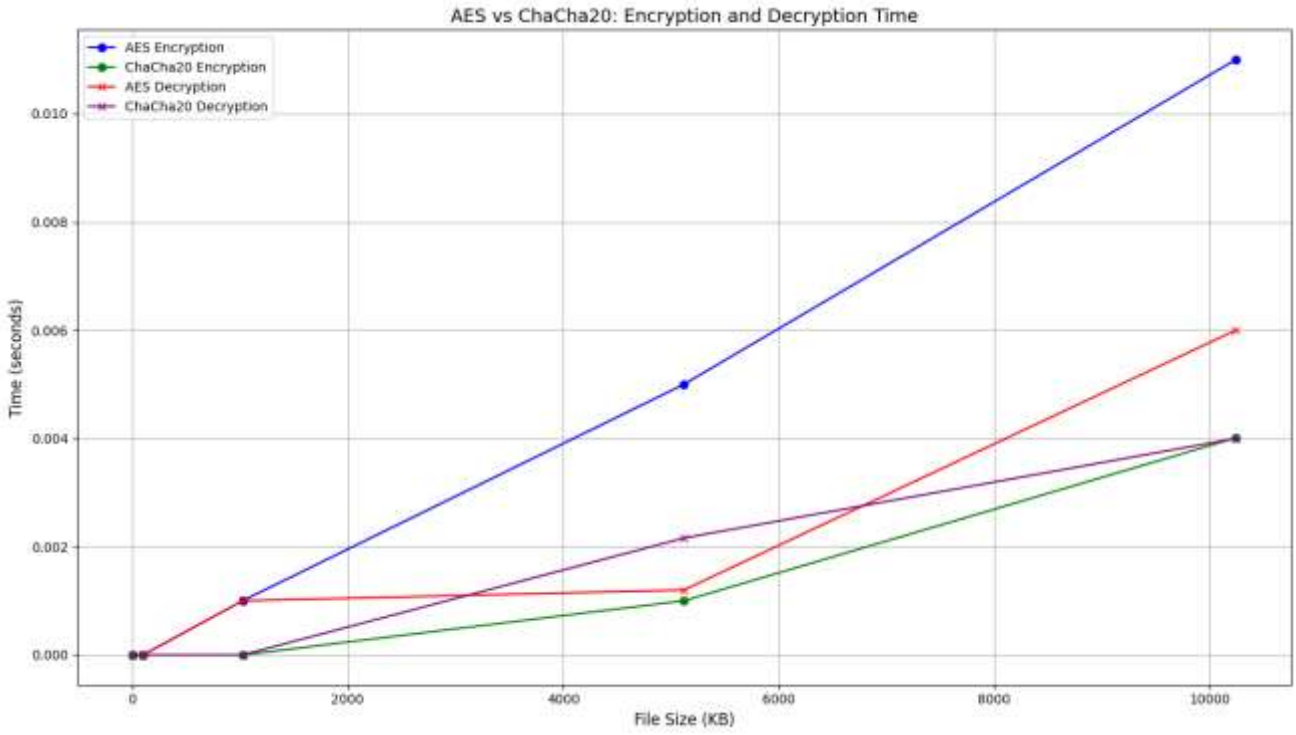
3.1. Encryption and Decryption Time Analysis

Figure 1 presents a comparative analysis of the encryption and decryption times of AES and ChaCha20 across increasing file sizes. As shown, the encryption time for AES increases significantly with file size, starting from 0.001 seconds at 1MB to 0.011 seconds at 10MB. In contrast, ChaCha20 demonstrates faster and more stable performance, with encryption times ranging from 0.001 seconds at 5MB to only 0.00401 seconds at 10MB.

Similarly, AES decryption time scales from 0.0012 seconds at 5MB to 0.006 seconds at 10MB, while ChaCha20 decryption time remains lower, peaking at 0.004 seconds for 10MB. These results indicate that ChaCha20 consistently outperforms AES in both operations.

Figure 1: AES vs ChaCha20 Encryption and Decryption Time across File Sizes
Figure 2: AES vs ChaCha20 Encryption and Decryption Time across File Sizes

Figure 3: AES vs ChaCha20 Encryption and Decryption Time across File Sizes



This performance disparity can be attributed to the structural differences between the algorithms. AES uses block-based encryption with padding and multiple transformation rounds, while ChaCha20 operates as a stream cipher with reduced overhead and fewer dependencies. This makes ChaCha20 more scalable and efficient for real-time encryption in low-resource systems.

3.2. Memory Usage Comparison

Table 1. shows the **peak memory consumption** of each algorithm during encryption and decryption. AES memory usage increased steadily from **0.614 MB (1KB file)** to **40.001 MB (10MB file)**. In comparison, ChaCha20 displayed much lower memory utilization, reaching only **20 MB** at 10MB input size.

Table 1: Peak Memory Usage Comparison of AES and ChaCha20

File Size (KB)	AES Peak Memory (MB)	ChaCha20 Peak Memory (MB)
1	0.614	0.002
100	0.392	0.196
1024 (1MB)	4.001	2
5120 (5MB)	20.001	10
10240 (10MB)	40.001	20

These findings highlight ChaCha20's superior memory efficiency, reinforcing its viability in constrained environments such as embedded systems, mobile applications, and IoT devices where RAM is limited.

3.3. Interpretation of Findings

The performance and memory efficiency of ChaCha20 confirm its reputation as a lightweight cryptographic algorithm suitable for modern security needs in constrained systems. These findings align with prior literature that supports the adoption of stream ciphers like ChaCha20 in environments where AES may be impractical due to its computational demands [3][4][5].

Despite AES's established strength in secure environments, its performance under CBC mode introduces delays that may be unacceptable in time-sensitive or resource-critical applications. Therefore, the experimental evidence supports the conclusion that ChaCha20 is a more suitable encryption method for use in resource-constrained environments.

3.4. Implications and Recommendations

The results imply that developers, engineers, and cybersecurity professionals working in IoT, edge computing, or low-power embedded systems should consider ChaCha20 as a default encryption standard where speed and memory usage are critical. AES may still be appropriate for high-security environments with hardware acceleration or legacy constraints, but its limitations in software-only implementations under constrained conditions must be acknowledged.

4. Conclusion

This study investigated and compared the performance of two widely used cryptographic algorithms—AES (Advanced Encryption Standard) and ChaCha20—with a specific focus on their applicability in resource-constrained environments. Using Python-based simulations, the research measured and analyzed the encryption time, decryption time, and peak memory consumption of each algorithm across varying file sizes from 1KB to 10MB.

The results revealed that ChaCha20 consistently outperformed AES in terms of both speed and memory efficiency. AES, while robust and secure, demonstrated increasing encryption and decryption times as file size grew, along with significantly higher memory usage. In contrast, ChaCha20 maintained faster execution and lower memory overhead, particularly at larger file sizes. These findings validate previous claims in the literature regarding ChaCha20's suitability for lightweight cryptographic applications, especially in IoT devices, embedded systems, and mobile platforms where computational resources are limited.

In conclusion, ChaCha20 proves to be a more efficient and scalable encryption algorithm under constrained conditions compared to AES. This study not only reinforces the growing relevance of lightweight cryptography but also provides practical insight for developers and system designers when choosing secure, yet resource-conscious, encryption solutions.

5. Appendix

5.1. Source Code

```
import os
import time
import psutil
import tracemalloc
import pandas as pd
```

```
import matplotlib.pyplot as plt
from cryptography.hazmat.primitives.ciphers import (
Cipher, algorithms, modes
)
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305

# === Utility Functions ===

def get_cpu_memory_usage():
process = psutil.Process(os.getpid())
cpu = psutil.cpu_percent(interval=0.1)
memory = process.memory_info().rss / (1024 * 1024)  # MB
return cpu, memory

def pad_data(data):
padder = padding.PKCS7(128).padder()
return padder.update(data) + padder.finalize()

def unpad_data(padded_data):
unpadder = padding.PKCS7(128).unpadder()
return unpadder.update(padded_data) + unpadder.finalize()

# === AES Simulation ===

def aes_encrypt_decrypt(data):
key = os.urandom(32)  # AES-256
iv = os.urandom(16)
backend = default_backend()
cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)

# Padding
padded_data = pad_data(data)

# Encryption
encryptor = cipher.encryptor()
start_enc = time.time()
encrypted = encryptor.update(padded_data) + encryptor.finalize()
end_enc = time.time()

# Decryption
decryptor = cipher.decryptor()
```

```
start_dec = time.time()
decrypted_padded = decryptor.update(encrypted) + decryptor.finalize()
end_dec = time.time()

decrypted = unpad_data(decrypted_padded)

return end_enc - start_enc, end_dec - start_dec

# === ChaCha20 Simulation ===

def chacha20_encrypt_decrypt(data):
key = ChaCha20Poly1305.generate_key()
nonce = os.urandom(12)
chacha = ChaCha20Poly1305(key)

# Encryption
start_enc = time.time()
encrypted = chacha.encrypt(nonce, data, None)
end_enc = time.time()

# Decryption
start_dec = time.time()
decrypted = chacha.decrypt(nonce, encrypted, None)
end_dec = time.time()

return end_enc - start_enc, end_dec - start_dec

# === Run Test ===

def run_simulation(file_size_kb=1024):
print(f"Testing with {file_size_kb} KB file...")
data = os.urandom(file_size_kb * 1024)

# AES
tracemalloc.start()
aes_enc, aes_dec = aes_encrypt_decrypt(data)
aes_memory = tracemalloc.get_traced_memory()[1] / (1024 * 1024)
tracemalloc.stop()

# ChaCha20
tracemalloc.start()
cha_enc, cha_dec = chacha20_encrypt_decrypt(data)
cha_memory = tracemalloc.get_traced_memory()[1] / (1024 * 1024)
```

```
tracemalloc.stop()

return {
"File Size (KB)": file_size_kb,
"AES Encryption Time (s)": round(aes_enc, 5),
"AES Decryption Time (s)": round(aes_dec, 5),
"AES Peak Memory (MB)": round(aes_memory, 3),
"ChaCha20 Encryption Time (s)": round(cha_enc, 5),
"ChaCha20 Decryption Time (s)": round(cha_dec, 5),
"ChaCha20 Peak Memory (MB)": round(cha_memory, 3)
}

# === Visualization Function ===

def plot_results(df):
plt.figure(figsize=(14, 8))
plt.plot(df["File Size (KB)"], df["AES Encryption Time (s)"], marker='o', label="AES Encryption", color="blue")
plt.plot(df["File Size (KB)"], df["ChaCha20 Encryption Time (s)"], marker='o', label="ChaCha20 Encryption", color="green")
plt.plot(df["File Size (KB)"], df["AES Decryption Time (s)"], marker='x', label="AES Decryption", color="red")
plt.plot(df["File Size (KB)"], df["ChaCha20 Decryption Time (s)"], marker='x', label="ChaCha20 Decryption", color="purple")

plt.xlabel("File Size (KB)", fontsize=12)
plt.ylabel("Time (seconds)", fontsize=12)
plt.title("AES vs ChaCha20: Encryption and Decryption Time", fontsize=14)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# === Main Execution ===

if __name__ == "__main__":
test_sizes = [1, 100, 1024, 5120, 10240]  # 1KB, 100KB, 1MB, 5MB, 10MB
results = []

for size in test_sizes:
result = run_simulation(size)
results.append(result)
```

```
df = pd.DataFrame(results)
df.to_csv("crypto_benchmark_results.csv", index=False)
print("\nBenchmark complete! Results saved to 'crypto_benchmark_results.csv'.")
print(df)

# Show chart
plot_results(df)
```

References

1. H. Mohammad and A. Abdullah, "Enhancement process of aes: a lightweight cryptography algorithm-aes for constrained devices", Telkomnika (Telecommunication Computing Electronics and Control), vol. 20, no. 3, p. 551, 2022. https://doi.org/10.12928/telkomnika.v20i3.23297
2. H. Madushan, I. Salam, & J. Alawatugoda, "A review of the nist lightweight cryptography finalists and their fault analyses", Electronics, vol. 11, no. 24, p. 4199, 2022. https://doi.org/10.3390/electronics11244199
3. I. Jomaa, R. Mohammed, N. Jaafar, & H. Ismael, "Cryptography by the echacha20 algorithm based on logistic and chirikov chaotic maps", Bilad Alrafidain Journal for Engineering Science and Technology, vol. 3, no. 1, p. 1-19, 2024. https://doi.org/10.56990/bajest/2024.030101
4. O. Jallouli, M. Chetto, & S. Assad, "Lightweight stream ciphers based on chaos for time and energy constrained iot applications",, p. 1-5, 2022. https://doi.org/10.1109/meco55406.2022.979708
5. S. Kumar, D. Kumar, R. Dangi, G. Choudhary, N. Dragoni, & I. You, "A review of lightweight security and privacy for resource-constrained iot devices", Computers Materials & Continua, vol. 78, no. 1, p. 31-63, 2024. https://doi.org/10.32604/cmc.2023.047084
6. A. Inoue, C. Guo, & K. Minematsu, "Nonce-misuse resilience of romulus-n and gift-cofb", Iet Information Security, vol. 17, no. 3, p. 468-484, 2023. https://doi.org/10.1049/ise2.12110
7. R. Hira, T. Kitahara, D. Miyahara, Y. Hara–Azumi, Y. Li, & K. Sakiyama, "Software evaluation for second round candidates in nist lightweight cryptography", Journal of Information Processing, vol. 31, no. 0, p. 205-219, 2023. https://doi.org/10.2197/ipsjjip.31.205
8. N. Li, Y. Ali, K. Hu, S. Nazir, & H. Zhao, "A hybrid mcdm approach of selecting lightweight cryptographic cipher based on iso and nist lightweight cryptography security requirements for internet of health things", Ieee Access, vol. 8, p. 220165-220187, 2020. https://doi.org/10.1109/access.2020.3041327
9. R. Muhammed, M. Al-Dabbas, & A. Alabaichi, "Review: dual method cryptography and steganography in video frames in iot", Iraqi Journal of Computer Communication Control and System Engineering, p. 139-155, 2023. https://doi.org/10.33103/uot.ijccce.23.3.12