Performance Comparison of AES and ChaCha20 over Local TCP Sockets

Michael Ballard Isaiah Silaen - 18223080
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 18223080@std.stei.itb.ac.id, silaenmichael@gmail.com

***Abstract*—Secure file transfer protocols require an efficient cryptographic algorithm to keep the integrity and confidentiality intact, and this is something that should be consistent for many devices with variation in hardware capabilities and capacities. This paper presents a comparative performance analysis AES-256-GCM (block cipher) and ChaCha20-Poly1305 (stream cipher) through a custom TCP (Transmission Control Protocol) application. The main focus of this paper is the design, implementation, and cross-platform experimental benchmarking of these two ciphers on two different ARM architectures, a modern flagship laptop processor (Apple Silicon M4 Pro) and a resource-limited mobile environment (Samsung Galaxy A70 via Android Termux). Benchmarks to measure parameters such as encryption and decryption times, CPU and memory consumption, and throughput were done using payloads ranging from 10 KB to 500 MB. The experimental framework uses chunked processing to ensure a stable memory allocation on the mobile phone's hardware. The results of the experiments demonstrate the differences in the architectural efficiency and performance of both ciphers under differing compute and thermal environments.**

***Keywords*—AES-256-GCM, ChaCha20-Poly1305, TCP Sockets, Performance Benchmarking, Authenticated Encryption, Cross-Platform Analysis**

I. Introduction

In the increasingly digitalized modern world, secure data transmission across networks is a requirement that is fundamental for any digital platform. As mobile computing becomes increasingly common and used everywhere, client-server applications must rely on cryptographic protocols that are robust to ensure that data confidentiality and integrity is still secured against security threats that are continuously evolving. However, the implementation of these encryption protocols carries a computational cost that can impact performance on a significant scale especially on mobile devices which are constrained by limited hardware resources and strict thermal limitations.

Two of the most prominent and commonly used authenticated encryption algorithms in modern communication are AES-256-GCM, an industry-standard block cipher [1] that combines the AES algorithm with a 256-bit key and the Galois/Counter Mode (GCM) operation mode [2], and ChaCha20-Poly1305, which is a stream cipher that pairs the ChaCha20 stream cipher with the Poly1305 Message Authentication Code for data confidentiality and integrity, with the ChaCha20-Poly1305 encryption algorithm being designed for high performance software execution [3]. While AES benefits heavily from dedicated cryptographic hardware acceleration on many modern architectures [6], ChaCha20 is built to operate efficiently using general-purpose processor instructions. This architectural difference shows a performance evaluation that must be considered and done when developing cross-platform applications.

In an effort to explore practical issues and methodologies in the development of applications that are secure in nature, this paper presents the design, implementation, and experimental testing of both of these ciphers. Specifically, this study analyzes and compares these cryptographic algorithms through a custom TCP file transfer experiment which benchmarks the performance of these two cryptographic algorithms across two different ARM architectures, a high-performance laptop processor (Apple Silicon M4 Pro) and a thermally-constrained mobile environment (Samsung Galaxy A70, driven by a Qualcomm Snapdragon 675 chipset). By measuring execution latency, network throughput, CPU utilization, and peak memory consumption across varying payload sizes, this research gives us an evaluation of how each cipher scales and performs under different computational environments.

To properly measure the computational reality of both ciphers, this study uses a custom client-server file transfer application built using standard TCP sockets. TCP here is a widely-used transport protocol that is used for reliable transmission [5]. A challenge in cross-architecture benchmarking, especially in this case where the experiment was done on a high-end laptop processor and an aged mobile processor which has a significant difference in capabilities and performance, is making sure that the testing environment does not introduce bias to the results through out-of-memory failures or aggressive garbage collection overhead. To mitigate this on the resource-limited Galaxy A70, the implementation of this experiment uses a 1 MB chunked processing methodology. The design in which the experiment was done ensures that peak memory consumption remains stable regardless of the payload size which in this study scales from 10 KB to 500 MB.

The performance of AES-256-GCM as defined in NIST Special Publication 800-38D and ChaCha20-Poly1305 as defined in RFC 8439 [4] is evaluated based on several core

hardware and software metrics. The benchmarks capture the raw encryption and decryption execution latency, the total network throughput (measured in MB/s), the average CPU utilization, and the peak memory usage. By measuring these specific variables and parameters, this experiment provides a clear insight into the architectural efficiency and the scaling characteristics of both ciphers across a hardware gap that is significant and generational.

II. Theoretical Background

A. Authenticated Encryption with Associated Data (AEAD)

Authenticated Encryption with Associated Data (AEAD) is a modern cryptographic tool designed to simultaneously guarantee the confidentiality, integrity, and authenticity of transmitted data. Historically, secure communication protocols treated encryption and authentication as separate operations. Systems would use one algorithm to encrypt the plaintext and use a separate cryptographic hash to verify it, but combining these things manually such as using the Encrypt-and-MAC systems often gives implementation flaws that are subtle but catastrophic which could lead to severe network vulnerabilities. AEAD resolves this issue by binding the encryption and authentication processes into a single cohesive and mathematically proven operation.

A standard AEAD scheme requires four primary inputs. The first of which being the secret key, which is the cryptographic key used to lock and unlock the data. The second being the Nonce/Initialization Vector (IV) which is a unique and arbitrary number only used once for any key to ensure that multiples of a same plaintext being encrypted will always result in a completely different ciphertext, and the thor being the plaintext message itself. The fourth input is the Associated Data 9AD), which represents contextual information that must be authenticated but exposed in plaintext such as IP addresses or TCP network headers. The AEAD algorithm processes these inputs to output the ciphertext alongside a fixed-length cryptographic verification code known as the Authentication Tag.

During the decryption phase, the receiving system recalculates the tag using the ciphertext and Associated Data. If even a single bit of the ciphertext or the Associated Data has been changed during the delivery process, the tags will mismatch and the algorithm will immediately reject the payload. In the context of this experiment, both the AES-256-GCM and ChaCha20-Poly1305 function as AEAD constructs, in which they ensure that any corrupted or altered file chunk sent across the TCP socket is instantly rejected before the system even starts processing it.

B. Advanced Encryption Standard in Galois/Counter Mode (AES-256-GCM)

The Advanced Encryption Standard (AES) is a symmetric block cipher established by the National Institute of Standards and Technology (NIST) as the definitive standard for securing sensitive data. In its AES-256 configuration, the algorithm uses a 256-bit cryptographic key to process data in fixed 128-bit blocks through 14 transformation rounds. However, block ciphers operating in traditional modes such as Cipher Block Chaining (CBC) often struggle with parallelization and lack native data authentication which leaves them vulnerable to tampering.

To resolve these limitations, AES is frequently used in Galois/Counter Mode (GCM). GCM transforms the foundational block cipher into a stream-oriented cipher by encrypting a mathematically incrementing counter value and XORing the resulting keystream with the plaintext. This approach makes the execution efficient and parallelized which is a big factor for maximizing network throughput in data-heavy TCP applications.

GCM also provides robust data authentication through the Galois Message Authentication Code (GMAC). GMAC computes a verification tag using polynomial multiplication over the Galois field $GF(2^{128})$ as the ciphertext is generated. This ensures that any illegal modification to the ciphertext is mathematically detectable.

AES-GCM benefits from modern systems architecture where there is dedicated cryptographic silicon. Most modern processors (like the ones used in this study) such as the Apple Silicon M4 Pro and the Qualcomm Snapdragon series of chips implement ARM cryptography extensions. These extensions provide hardware-level instructions for AES matrix transformations and Galois field multiplication which allows AES-256-GCM to achieve high throughputs while minimizing CPU utilization.

C. ChaCha20-Poly1305 Stream Cipher

ChaCha20-Ply1305 Stream Cipher is a high-speed stream cipher developed by Daniel J. Bernstein which has emerged as a widely adopted software-optimized alternative to AES. When combined with the Poly1305 authenticator, it forms a highly secure AEAD construct standardized by the Internet Engineering Task Force (IETF), and unlike AES which operates on discrete data blocks, ChaCha20 generates a continuous keystream that is XORed with the plaintext.

The fundamental architectural advantage of ChaCha20 lies in its reliance on ARX operations (Addition, Rotation, XOR). The algorithm initializes a 512-bit state (represented as a 4x4 matrix of 32-bit words), using the 256-bit key, a 96-bit nonce, and a block counter. It then applies 20 rounds of ARX transformations to diffuse the state. Because ARX operations execute natively and efficiently on almost all general-purpose Arithmetic Logic Units (ALUs), ChaCha20 has a high cryptographic throughput purely in software and bypassing th need for specialized hardware acceleration.

The ARX structure also protects the cipher against cache-timing side-channel attacks, a vulnerability that often appears on software-only implementations of AES. To satisfy the AEAD scheme, the ciphertext is processed through Poly1305 (a high speed cryptographic Message Authentication Code (MAC) function) that provides integrity and authenticity verification for both the ciphertext and Associated Data before the plaintext is accepted..

In the context of this study, ChaCha20-Poly1305 is particularly important for resource constrained mobile environments. It ensures that devices lacking dedicated AES hardware acceleration or those operating in strict thermal

conditions can still maintain secure and high-bandwidth connections without overloading the central processor.

D. ARM Architecture and Cryptographic Hardware Acceleration

Both the Apple Silicon M4 Pro and the Qualcomm Snapdragon 675 are built on the Advanced RISC Machine (ARM) instruction set architecture. A feature of modern ARM processors is the inclusion of the ARM Cryptography Extensions (CE). These hardware-level instructions are designed to accelerate symmetric cryptographic primitives which makes algorithms like the AES-256-GCM to have a significant rise in throughput gains and reduced power consumption compared to pure software execution. Meanwhile, ChaCha20-Poly1305 does not use the cryptographic extensions but instead relied on ARM's Advanced SIMD (NEON) architecture which heavily parallelizes the general-purpose ARX operations.

Even though both processors in this study supports AES acceleration, their operational realities have a significant difference. The M4 Pro is an ARMv9 desktop architecture featuring massive instruction-level parallelism, high-bandwidth memory, and active cooling. Meanwhile, the Snapdragon 675 is an older ARMv8 mobile System-on-Chip (SoC) operating within strictly constrained thermal limits.

E. Transmission Control Protocol (TCP) and Socket Communication

The Transmission Control Protocol (TCP) provides reliable, ordered, and error-checked delivery of byte streams between communicating applications. Within the custom secure file-transfer framework, TCP acts as the transport layer for the encrypted data transmission. Because the algorithms (AES-256-GCM and ChaCha20-Poly1305) verify ciphertext integrity before decryption, any corruption, truncation, or reordering of the received ciphertext will fail the authentication. TCP's reliable and ordered delivery ensures that the encrypted byte stream is reconstructed correctly before decryption.

The implementation uses a standard socket-based communication and application-layer framing to transmit large files efficiently. Instead of loading an entire 500 MB file into memory for example, the sender divides the payload into discrete 1 MB chunks. Each chunk is encrypted independently and prefixed with a 4-byte length header before being sent over the TCP connection. When received, the length header enables the receiver to know the boundaries of the ciphertext and rebuild the original data stream. This particular design limits peak memory usage while ensuring reliable transfer of files up to 500 MB in size.

III. Methodology and Implementation

A. Experimental Environment and System Architecture Overview

The experimental framework is designed to evaluate the cryptographic performance of AES-256-GCM and ChaCha20-Poly1305 across two very different compute environments. The primary high-performance testbed is an Apple MacBook equipped with the M4 Pro ARMv9 architecture. This environment is a representation of a modern laptop workload capable of active cooling, high-bandwidth memory, and advanced hardware-level cryptographic acceleration.

Meanwhile, the resource-constrained testbed is a mobile platform in the form of the Samsung Galaxy A70 smartphone powered by the Qualcomm Snapdragon 675 processor using ARMv8 architecture. Since the experimental framework is all in Python, a Termux terminal emulator is installed and used on the phone. The mobile device operates under passive cooling which makes it prone to thermal throttling under heavy workloads.

To erase the unpredictable latency and bandwidth fluctuations usually seen with physical Wi-Fi or cellular networks, the client-service architecture is done entirely over the local loopback interface (127.0.0.1) on each device, This architectural design ensures that the measured throughput directly represents the maximum cryptographic processing capability of the CPU and memory subsystems rather than external network hardware limitations. The core app itself is developed in Python using the standard POSIX [socket] library for TCP transmission and the [cryptography] package to handle AES and ChaCha20.

B. Cryptographic Algorithm Implementation

To ensure cryptographic integrity and accurately capture underlying hardware optimizations, the encryption primitives were implemented using the hazmat (Hazardous Material) module of Python's cryptography library. This study uses this specific library because custom and manual implementations of cryptographic maths are prone to vulnerabilities and performance degradation, meanwhile this library ensures that benchmarks reflect standard performance.

The implementation uses an object-oriented architecture to abstract the specific cipher logic away from the networking layer. A base StreamAEAD class is defined to manage the cryptographic stage during the chunked file transfer.

```
class StreamAEAD:
    """Base class for handling chunked AEAD
encryption/decryption."""
    def __init__(self, key: bytes, base_nonce:
bytes):
        self.key = key
        self.base_nonce = base_nonce
    def _get_nonce(self, chunk_index: int) ->
bytes:
        base_int =
int.from_bytes(self.base_nonce, 'big')
        return (base_int ^
chunk_index).to_bytes(12, 'big')
```

Because the application segments the payload into discrete 1 MB chunks to manage memory, generating a unique Nonce/Initialization Vector (IV) for every single chunk is a

strict cryptographic requirement, reusing a nonce with the same key in either GCM or Poly1305 compromises the cipher. The _get_nonce method safely derives a unique 12-byte nonce per chunk by applying a bitwise XOR operation between the randomly generated base session nonce and the incrementally increasing chunk_index.

Child classes (AES256GCM_Cipher and ChaCha20_Cipher) inherits this strict nonce derivation logic. They wrap the encrypt and decrypt functions and therefore allowing the benchmarking orchestrator to swap algorithms for testing without altering the core TCP transmission logic.

```
class AES256GCM_Cipher(StreamAEAD):
    def __init__(self, key: bytes, base_nonce:
bytes):
        super().__init__(key, base_nonce)
        self.cipher = AESGCM(key)
    def process(self, data: bytes, index: int,
encrypt=True) -> bytes:
        nonce = self._get_nonce(index)
        if encrypt:
            return self.cipher.encrypt(nonce,
data, None)
        return self.cipher.decrypt(nonce, data,
None)

class ChaCha20_Cipher(StreamAEAD):
    def __init__(self, key: bytes, base_nonce:
bytes):
        super().__init__(key, base_nonce)
        self.cipher = ChaCha20Poly1305(key)
    def process(self, data: bytes, index: int,
encrypt=True) -> bytes:
        nonce = self._get_nonce(index)
        if encrypt:
            return self.cipher.encrypt(nonce,
data, None)
        return self.cipher.decrypt(nonce, data,
None)
```

The object-oriented design standardizes the cryptographic interface through the unified process method. By encapsulating the underlying encrypt and decrypt functions into a single callable signature, the network layer becomes entirely agnostic to the used cipher. The third parameter in underlying library calls (passed here as None) represents the optional Associated Data, but for this raw throughput benchmark the integrity check focuses on the ciphertext payload.

C. Network Framing and Data Chunking Mechanism

Unlike UDP (User Datagram Protocol), TCP is a continuous stream protocol that lacks inherent message boundaries, and this is a structural challenge for AEAD algorithms. Because GCM and Poly1305 must process the exact ciphertext to successfully validate the authentication tag, a partial read from the TCP buffer would alter the data structure and make it fail the cryptographic verification. To deal with this, a custom Length-Value framing protocol was implemented at the application layer to manage the TCP socket transmission.

```
import socket

def send_msg(sock: socket.socket, data: bytes):
    """Prefixes data with a 4-byte length header
and sends it over TCP."""
    length_prefix = len(data).to_bytes(4, 'big')
    sock.sendall(length_prefix + data)

def recvall(sock: socket.socket, n: int) ->
bytes:
    """Helper to reliably receive exactly n
bytes."""
    data = bytearray()
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet:
            return None
        data.extend(packet)
    return bytes(data)

def recv_msg(sock: socket.socket) -> bytes:
    """Reads the 4-byte length header, then
reads the payload."""
    raw_msglen = recvall(sock, 4)
    if not raw_msglen:
        return None
    msglen = int.from_bytes(raw_msglen, 'big')
    return recvall(sock, msglen)
```

As shown, the send_msg function prefixes every encrypted chunk with a 4-byte header representing the exact byte length of the payload. On the receiving end, the recv_msg function intercepts this header first. The application determines the precise size of the incoming ciphertext and uses the recvall helper function to poll the TCP socket in a while loop until the complete chunk is reconstructed accurately by decoding the 4-byte integer.

This framing mechanism also facilitates the 1MB data chunking tactic, because rather than attempting to load and encrypt a massive 500 MB file into RAM which would trigger Out-Of-Memory exceptions and aggressive garbage collection on the Android testbed, the system processes the file sequentially. This strategy ensures that the peak memory footprint is still bounded and isolates the cryptographic throughput from OS-level memory constraints.

D. Resource Monitoring and Profiling Methodology

To accurately quantify the computational cost of each cryptographic algorithm, the system must note CPU utilization and memory consumption without degrading the data transfer's throughput. A dedicated ResourceMonitor class was implemented using Python's threading and psutil modules.

The monitor samples hardware metric every 50 milliseconds without interrupting the TCP socket operations by executing the logic on a separate background thread.

A challenge in cross-architecture profiling especially on modern mobile operating systems is process sandboxing, therefore the profiling implementation was specifically adapted to ensure stability across both the MacOS and Android testbeds.

```
class ResourceMonitor:
    """Runs a background thread to poll CPU and
Memory usage asynchronously."""
    def __init__(self):
        self.keep_running = True
        self.cpu_usages = []
        self.mem_usages = []
        self.process = psutil.Process()
        try:

self.process.cpu_percent(interval=None)
        except psutil.AccessDenied:
            pass
        self.thread =
threading.Thread(target=self._monitor,
daemon=True)

    def start(self):
        self.thread.start()

    def _monitor(self):
        while self.keep_running:
            try:
                cpu =
self.process.cpu_percent(interval=None)
                self.cpu_usages.append(cpu)
            except psutil.AccessDenied:
                self.cpu_usages.append(0.0)
            except Exception:
                pass

            try:

self.mem_usages.append(self.process.memory_info(
).rss / (1024 * 1024))
            except Exception:
                pass
            time.sleep(0.05)

    def stop(self) -> tuple:
        self.keep_running = False
        self.thread.join()
        avg_cpu = sum(self.cpu_usages) /
len(self.cpu_usages) if self.cpu_usages else 0.0
        peak_mem = max(self.mem_usages) if
self.mem_usages else 0.0
        return avg_cpu, peak_mem
```

As seen in the _monitor method, the script queries the active Python process for its CPU percentage and Resident Set Size (RSS) memory footprint using psutil. Because access to process statistics can vary across operating systems and execution environments, the implementation explicitly handles psutil.AccessDenied exceptions. If a CPU utilization query is denied, the monitor records a value of 0.0 and continues execution without interruption. Memory measurements are collected independently through separate exception handling logic, and after the file transfer is done, the monitoring thread is terminated and the collected samples are processed to compute the average CPU utilization and peak RSS memory consumption seen during the cryptographic operation.

E. Benchmarking Procedure and Test Parameters

To comprehensively evaluate the scaling characteristics of the cryptographic algorithms, the experimental procedure was designed to simulate both discrete messaging and sustained high-volume data transfers. The test suite uses a tiered payload structure (10 KB, 1 MB, 50 MB, 200 MB, 500 MB files). These test files were generated directly on the internal storage of the host devices before execution to prevent storage I/O caching mechanisms from damaging the cryptographic timings.

The automated benchmarking orchestrator was implemented to systematically cycle through testing and ensuring identical testing conditions for both ciphers, and to print results into a csv file.

```
def main():
    sizes_mb = [0.01, 1, 50, 200, 500]
    ciphers = ["AES256GCM", "CHACHA20"]
    results = []

    for size in sizes_mb:
        filepath = os.path.join(config.DATA_DIR,
f"test_{size}MB.bin")
        if size < 1:
            with open(filepath, 'wb') as f:
                f.write(os.urandom(int(size *
1024 * 1024)))
        else:
            if not os.path.exists(filepath):
                generate_dummy_file(filepath,
int(size))

        for cipher in ciphers:
            print(f"Running benchmark: {cipher}
| {size} MB")
            metrics = send_file(filepath,
cipher)
            results.append(metrics)
            time.sleep(1)

    csv_file = "benchmark_results.csv"
    keys = results[0].keys()
    with open(csv_file, 'w', newline='') as f:
        dict_writer = csv.DictWriter(f,
fieldnames=keys)
        dict_writer.writeheader()
        dict_writer.writerows(results)
```

```
    print(f"\nBenchmarks complete! Results saved
to {csv_file}")
```

As seen in the orchestrator loop, the script sequentially executes the transfer for each file size using AES-256-GCM and then ChaCha20-Poly1305. To capture the highest resolution timing data possible, the underlying send_file function measures execution latency using Python's time.perf_counter().

```
 total_time = time.perf_counter() -
total_start_time
        cpu_avg, mem_peak =
client_monitor.stop()

        resp_bytes = net_mgr.recv_msg(sock)
        server_metrics =
json.loads(resp_bytes.decode('utf-8'))

    throughput = (filesize / (1024 * 1024)) /
total_time
```

IV. Results and Discussions

A primary objective of the experimental design was to ensure that the cross-platform benchmarking suite can handle massive payloads (up to 500 MB) without having Out-Of-Memory exceptions on the resource-constrained mobile testbed. The collected telemetry data validates the 1 MB Length-Value (LV) chunking and framing methodology.

TABLE I. Cryptographic Performance Benchmarks - macOS (M4 Pro)

Cipher	*Payload (MB)*	*Total Time (s)*	*Throughput (MB/s)*	*Avg. CPU (%)*	*Peak Memory (MB)*
AES-256-GCM	0.01	0.0001	126.04	94.8	23.3
ChaCha20	0.01	0.0001	72.37	69.6	23.4
AES-256-GCM	1.0	0.0014	706.46	75.8	25.7
ChaCha20	1.0	0.0032	311.1	92.5	29.0
AES-256-GCM	50.0	0.0329	1521.84	59.4	37.1
ChaCha20	50.0	0.0549	911.36	66.3	48.4
AES-256-GCM	200.0	0.1316	1519.77	47.3	55.7
ChaCha20	200.0	0.1932	1035.26	61.3	55.7
AES-256-GCM	500.0	0.3335	1499.07	41.3	61.9
ChaCha20	500.0	0.4858	1029.19	59.5	61.9

TABLE II. Cryptographic Performance Benchmarks - Android (Snapdragon 675)

Cipher	*Payload (MB)*	*Total Time (s)*	*Throughput (MB/s)*	*Avg. CPU (%)*	*Peak Memory (MB)*
AES-256-GCM	0.01	0.0001	9.62	173.7	23.1
ChaCha20	0.01	0.0043	2.34	203.5	23.1
AES-256-GCM	1.0	0.0251	39.91	0.0	23.2
ChaCha20	1.0	0.0139	71.75	112.8	26.0
AES-256-GCM	50.0	0.2807	178.11	60.4	26.0
ChaCha20	50.0	0.3816	131.03	57.9	26.0
AES-256-GCM	200.0	1.8157	110.15	43.8	26.0
ChaCha20	200.0	5.8606	34.13	19.4	26.0
AES-256-GCM	500.0	15.2052	32.88	18.4	26.0
ChaCha20	500.0	16.3387	30.6	19.0	22.9

Across all test iterations on the Termux Galaxy A70, peak memory footprint remained stable. During the AES-256-GCM 50 MB payload transmission, the client memory peaked at 26.0 MB, and when the payload was increased tenfold to 500 MB, peak memory footprint still remained at 26.0 MB. Stability was also there for the ChaCha20-Poly1305 runs, with a peak of 26.0 MB for the 200 MB payload. On the M4 Pro MacOS testbed, which features a superior physical RAM and chip, the memory is still very optimized, with peak memory footprint at 61.9 MB for both AES-256-GCM and ChaCha20-Poly1305.

One of the most interesting observations can be seen in the experimental dataset during sustained processing on the mobile testbed as there is a sudden collapse in throughput as the payload scales. From Table II, the Android testbed handles the 50 MB payload with ease (AES-256-GCM has a throughput of 178.11 MB/s while using 60.4% of the CPU, and ChaCha20-Poly1305 reaches throughput of 131.03 MB/s while using 57.9% of the CPU). However, when payload scales to 200 MB and 500 MB, performance degrades massively (at the 500 MB mark, AES-256-GCM throughput collapses to 32.88 MB/s and ChaCha20 to 30.60 MB/s). From Table II we can also see that memory consumption is still stable at 26.0 MB, therefore this performance collapse couldn't have been caused by things like software bottlenecks or garbage collection overhead. Instead, this may show the effects of thermal throttling, and under sustained cryptographic execution, the Snapdragon 675 reaches its limit. To prevent damage, the Android kernel downclocks the processor therefore lowering the cryptographic throughput to manage heat. Meanwhile, from Table I it can be seen that the M4 Pro has an almost perfect linear scaling supported by modern architecture and active cooling. AES-256-GCM throughput is 1521.84 MB/s at a 50 MB payload and 1499.07 MB/s at 500 MB.

The comparative data gives an insight at the compute efficiency gap between hardware-accelerated block ciphers

and software-optimized stream ciphers. The benefits of dedicated ARM cryptography extensions is undoubted on the M4 Pro architecture, as during the 500 MB payload transfer, AES-256-GCM has a throughput of 1499.07 MB/s while consuming only 41.3% of the average CPU, meanwhile ChaCha20-Poly1305 processed the same payload at a slower 1029.19 MB/s while requiring a higher CPU use of 59.5%. This shows that when acceleration is available, AES provides a superior ratio of throughput to overhead.

Meanwhile, the Snapdragon 675 benchmarking highlights the efficiency of ChaCha20's ARX design. Before thermal throttling collapses performance, ChaCha20 maintains a 131.03 MB/s throughput while using 57.9% of the CPU on the 50 MB payload run. While AES-256-GCM still outperforms it at 178.11 MB/s throughput at 60.4% CPU usage, the performance difference on the older mobile architecture is noticeably smaller than on the M4 Pro testbed. And during the heavy throttling that occurs on the 500 MB payload run, the throughput of both ciphers are almost at the same level (32.88 MB/s for AES and 30.60 for ChaCha20). This shows that under heavy hardware pressure, the physical limits of the silicon is the bottleneck and therefore erasing the theoretical advantage of cryptographic extensions.

V. Conclusion

This study presented an evaluation of two widely-used authenticated encryption algorithms (AES-256-GCM and ChaCha20-Poly1305) by measuring their performance across a modern laptop processor (Apple M4 Pro) and a resource-constrained mobile device (Galaxy A70, Snapdragon 675). To ensure a valid comparison that isolates cryptographic execution from operating system memory limits, a custom TCP client-server application was built using a 1 MB Length-Value chunking mechanism. The telemetry data confirmed that this framing methodology successfully managed peak memory consumption to a maximum of 26.0 MB on the mobile testbed and prevented Out-Of-Memory exceptions even when transmitting 500 MB payloads.

The experimental results show that hardware acceleration is still the decisive factor in cryptographic efficiency. On the M4 Pro environment it can be seen that AES-256-GCM still consistently delivers higher throughput with lower CPU utilization which highlights the superiority of dedicated ARM cryptography extensions. But, the software-optimized ChaCha20 algorithm still proved to be competitive especially on the mobile architecture before hardware fatigue sets in.

The mobile benchmarks shows the impact of thermal throttling on sustained operations, therefore regardless of the ciphers' efficiency, continuous processing of large payloads on non-actively cooled devices resulted in a major collapse in throughput, showing that actual mobile cryptographic performance is still bound by physical thermal envelopes. For modern systems, this study uncovers a possible optimal approach where AES-256-GCM is prioritized when hardware acceleration is available, while defaulting to ChaCha20-Poly1305 to ensure secure communication on highly constrained devices.

VI. Acknowledgment

The author expresses gratitude and sincere thanks to Dr. Ir. Rinaldi Munir, M.T. for being an exceptional instructor and providing guidance throughout the course of the II4021 Cryptography class with great care and passion.

VII. Appendix

The exact and complete source code that was used to run the benchmark tests could be found at https://github.com/Michael-Ballard-Isaiah-Silaen/ChaCha20-vs-AES.git.

References

[1] J. Daemen and V. Rijmen, The Design of Rijndael: AES—The Advanced Encryption Standard. Berlin, Germany: Springer-Verlag, 2002.

[2] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," NIST Special Publication 800-38D, Nov. 2007.

[3] D. J. Bernstein, "ChaCha, a variant of Salsa20," Workshop Record of SASC 2008: The State of the Art of Stream Ciphers, Lausanne, Switzerland, pp. 3–5, 2008.

[4] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," RFC 8439, Internet Engineering Task Force, Jun. 2018.

[5] W. Eddy, Ed., "Transmission Control Protocol (TCP)," RFC 9293, Internet Engineering Task Force, Aug. 2022.

[6] S. Gueron, "Intel Advanced Encryption Standard (AES) New Instructions Set," Intel Corporation, White Paper, 2010.

DECLARATION

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Bandung, 19 Juni 2026

Michael Ballard Isaiah Silaen, 18223080