Bachelor's Thesis
(Course Code: XB_40001)

Design and Evaluation of a Secure Peer-to-Peer File Sharing Program for Resource-Constrained Devices

by

Cesar Guillen Cuñat
(Student Number: 2760847)

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

June 23, 2025

Certified by .....
Sabine Oechsner
Assistant Professor
First Supervisor

Certified by .....
Atze van der Ploeg
Assistant Professor
Second Reader

Design and Evaluation of a Secure Peer-to-Peer File Sharing Program for Resource-Constrained Devices

Cesar Guillen Cuñat
Vrije Universiteit Amsterdam
Amsterdam, NL
c.guillen.cunat@student.vu.nl

ABSTRACT

This thesis presents the design, development, and evaluation of a secure peer-to-peer file sharing program for embedded systems. The program uses a WiFi network to allow peers to download files from each other in a secure manner, even in the presence of an attacker on the network. The program takes into consideration attacks such as man-in-the-middle attacks, replay attacks, IP spoofing, etc. This thesis also explores different implementations of this program with each configuration using different popular lightweight encryption schemes, which are evaluated based on parameters such as; memory usage, time efficiency, etc.

The findings highlight the trade-offs involved in implementing secure communication protocols on resource-constrained embedded devices, particularly in terms of performance, memory usage, and design complexity.

1 INTRODUCTION

The number of IoT (Internet of Things) devices has increased by 15% in 2022 and is expected to reach 40 billion by 2030 [33]. These devices have the ability to communicate with each other or with a centralized server to send personal or sensitive information, therefore, it is important for these devices to use standardized, modern, and safe cryptography to keep private data safe in the presence of an attacker. Due to the nature of these devices, they are not capable of handling conventional cryptography schemes, as they have limited memory and processing power. To address these challenges, lightweight cryptographic libraries and functions have been developed to provide these devices with the tools needed to perform these tasks safely with the limited resources on which these embedded systems operate with.

Although these solutions allow for secure data exchange, secure peer-to-peer (P2P) file sharing on resource-constrained devices, particularly for large files in hazardous and highly unpredictable network environments, remains underexplored. Existing P2P systems often rely heavily on significantly more powerful hardware, leaving a notable gap in efficient, secure file-sharing solutions specifically tailored for embedded systems.

This thesis addresses this gap by designing and evaluating a secure P2P file-sharing program for embedded systems, capable of reliably operating in environments where attackers may attempt to compromise the system. This project takes an extreme approach, explicitly focusing on sharing large files to highlight the performance and security challenges of lightweight cryptography. This thesis investigates the following research question: How can a secure peer-to-peer file-sharing program be designed and evaluated to achieve efficient and robust performance on resource-constrained devices under a realistic threat model?

2 BACKGROUND

2.1 Peer-to-Peer File Sharing

One of the most common networking architectures is the client-server model, where a single central server is responsible for providing resources and services to connected clients. On the other hand, peer-to-peer (P2P) networking is a distributed architecture where each peer acts as both a client and a server [37]. The main advantages of this architecture choice are the ability to communicate directly to other peers without having to hop through a central server first. It also has excellent scalability. In the context of this project, it means that there is no need another microcontroller acting as a server. This reduced costs and decreased the complexity of the implementation, as it narrowed the possible attack paths that could be used to compromise the integrity of the system. Using this P2P approach results in the file contents being sent directly to the client, which reduces the network load.

In the context of this project, file sharing refers to the ability for one peer to download a file that is being hosted by another peer. For example, Alice can connect to Bob's server, Alice can then issue a command which will download any file that is stored on Bob's file system. Examples of similar file-sharing implementations include and SMB [32].

2.2 Cryptography Purpose & Functions

As Ronald Rivest explains, "Cryptology provides methods that enable a communicating party to develop trust that his communications have the desired properties, despite the best efforts of an untrusted party."[31]. Cryptography has been seamlessly integrated into our modern world, where many devices have to communicate with each other in a secure manner to prevent attackers from compromising communications and systems. Most people are unaware of all the complex algorithms or network protocols that are in place today. They are flawlessly integrated into our communication systems and provide us with three extremely important properties.

2.2.1 Confidentiality. Refers to the transfer of information, files, or data in a way that only authorized individuals are allowed to read and access it [26]. To achieve confidentiality, encryption functions can be used to convert the input, called the plaintext, into the final output, called the ciphertext. This ciphertext is the encrypted plaintext message. An attacker reading the ciphertext would not be able to decipher the contents and derive the plaintext message. To be able to decipher the ciphertext, a cryptographic key is needed which can be used to decrypt the ciphertext back into its corresponding plaintext.

A very rudimentary cipher is the Caeser cipher, which was used by Roman generals to send encrypted messages to other high-ranking officials, even if the message was captured, the enemies would not be able to decipher its contents. The encryption scheme involved shifting each letter of the plaintext by n positions within the alphabet. For example, with a shift of $n = 3$, the letter 'A' becomes 'D', 'B' becomes 'E', and so on. Figure 1 shows this encryption process. Although it offers minimal security by modern standards, it serves to illustrate the concept of confidentiality [35].

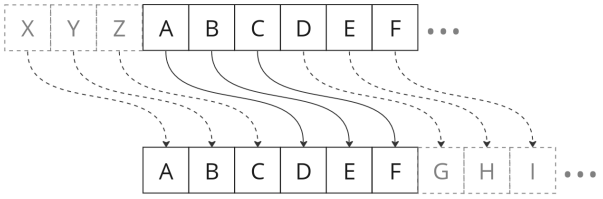


Figure 1: *Caesar cipher with a shift of n = 3.*

Encryption is necessary in scenarios where messages have to be kept secret from non-authorized individuals. Just as people whisper to share secrets in a crowded room, encryption ensures that sensitive information is not exposed to unintended recipients. We would not like to scream private information in a room full of people. Without encryption, messages are sent in plaintext, and the internet is like a room full of people, it can be assumed that anyone could be listening. An attacker could steal passwords, personal data, and other sensitive information.

2.2.2 Authentication. Enables systems to confirm that the sender of a message is genuinely who they claim to be, the sender and the receiver must both be authenticated before sending and receiving data [20]. Most modern cryptographic algorithms offer support for Authenticated Encryption (AE), which simultaneously provides confidentiality, message integrity, and authenticity in a single step. A generalization of this concept is Authenticated Encryption with Associated Data (AEAD). The Associated Data field (AD) includes data that are not encrypted, but it is still verified during decryption to verify that it was not tampered with, for this reason it is mainly used in metadata fields that need to be in plaintext but still have to be verified. during the decryption of the entire payload.

By including authentication as part of the encryption operation itself, AE and AEAD result in any tampering with the message or associated data making decryption fail, providing strong protection against forgery and tampering.

2.2.3 Integrity. Refers to not being able to modify data without having the appropriate permission to do so. This implies that the message received is exactly the same as the one sent, without being manipulated while it was in transit or kept in storage [26]. This property is achieved by using hash functions.

A hash function takes an arbitrarily long string as input and outputs a fixed length string [34]. Hash functions are deterministic in nature; the same input will always produce the same output, but a slight modification to the input will produce a completely different hash value. Modern hash functions have security properties such as collision resistance which states that it is computationally infeasible to find two strings that produce the same hash value [1]. This means that an attacker should not be able to find a different string from the original message that produces the same hash value. Bypassing this integrity check can lead to the person receiving this malicious message to be tricked into performing an action that compromises their system or leaks sensitive information.

Integrity can also be verified using AEAD which will output an error during the decryption process if the message has been tampered with.

2.3 Key Exchange Protocols

To decrypt a ciphertext back into its corresponding plaintext a cryptographic key is needed, this key must be the same for the sender and the receiver. Key-exchange protocols are essential for establishing a shared secret between two parties over an insecure channel. These key exchange must happen with the premise that an attacker will be listening and reading all data sent between the two parties; therefore, key exchange protocols must make sure that even if an attacker observes all transmitted messages, they cannot deduce the resulting shared key.

One of the first and most common key exchange methods is the Diffie–Hellman key exchange protocol, which was proposed in 1976 by Whitfield Diffie and Martin Hellman. The protocol was a milestone in cryptography, as it allowed two communicating parties to establish a shared secret without previously exchanging private information. Even if the attacker intercepted all the messages he or she would not be able to derive the shared secret [24].

2.4 Nonces

A nonce (number once) is a random or sequential number added to a cryptographic operation so that each message is unique. Its main purpose is to prevent replay attacks, where an attacker intercepts and resends a valid encrypted message to attempt to trick a system into performing the same operation over and over again. By incorporating a nonce, encryption algorithms ensure that even if the same plaintext is encrypted several times, the resulting ciphertext will never be the same, making the communication both unique and more secure [22].

2.5 Lightweight Cryptography

Lightweight cryptography refers to cryptographic algorithms specifically designed to operate efficiently on resource-constrained devices, such as microcontrollers. The aim of lightweight cryptography is to provide embedded systems with a security solution while using less memory, computing power, and energy [8]. The project focuses on three well-known symmetric encryption algorithms, ChaCha20, Ascon, and AES. To implement these encryption algorithms into the project, lightweight cryptographic libraries were used, as they provide a collection of cryptographic primitives.

2.5.1 Lightweight Cryptographic Libraries. In order to ease the use of lightweight cryptographic algorithms to real-world uses cases, various libraries have been developed that run the algorithms in an efficient and portable way. These libraries are optimized for low-resource environments and have secure, small, and

high-performance implementations of encryption, hash, and authentication operations. For this project, two libraries were used, MbedTLS and WolfSSL. Their differences and applications are discussed below.

2.5.2 Mbed TLS. An open-source cryptographic library primarily designed for embedded systems, offering implementations of optimized cryptographic primitives for resource-constrained environments such as microcontrollers [25]. It is modular and lightweight, making it suitable for developers who need fine-grained control over which cryptographic functions to include in their projects.

For this project, MbedTLS was used to obtain the SHA256 hash of files. The project also made use of their ChaCha20 encryption function which was used to encrypt files.

2.5.3 Wolf SSL. A small, portable, embedded SSL/TLS library targeted for use by embedded systems developers. It is an open-source implementation of TLS written in the C programming language [38, 39]. Similar to Mbed TLS it contains lightweight implementations for cryptographic functions, making it ideal for Iot devices.

Both of these libraries offers a wide selection of cryptographic primitives (e.g., AES, SHA, RSA) optimized for performance on microcontrollers.

For this project, WolfSSL was used in a similar way to MbedTLS to obtain the SHA256 hash of the files. The project also makes use of WolfSSL's AES-CBC and AES-GCM functions to encrypt messages and files.

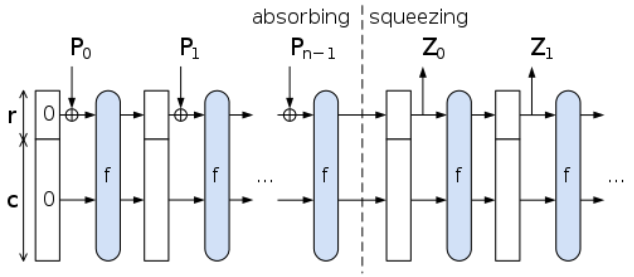


Figure 2: ***The sponge construction for hash functions. Pi are blocks of the input string, Zi are hashed output blocks.***

2.6 Ascon, ChaCha20 & AES

2.6.1 Ascon. Ascon is a family of lightweight cryptographic algorithms specifically designed for authenticated encryption and hashing in resource-constrained environments, such as IoT devices and embedded systems [13]. Ascon was selected as new standard for lightweight authenticated encryption and hashing in the NIST Lightweight Cryptography competition (2019–2023) [14]. Unlike general-purpose libraries, Ascon is not a full cryptographic suite, it is a specific family of lightweight cryptographic algorithms that are based on duplex construction [13], which internally uses permutation-based transformations [5]., which in simple terms is a function that rearranges or transforms input bits into a complex but deterministic string such that it is invertible.

Ascon also introduces its own hash function, Ascon-Hash which uses the sponge construction [4]. It produces a hash string of 256 bits in length or 32 bytes. Figure 2 shows how the sponge construction "absorbs" the input string into a fixed output length string.

2.6.2 ChaCha20. ChaCha20 is a 256-bit lightweight stream cipher designed by D. J. Bernstein. ChaCha20 is a refinement of the Salsa20 algorithm. An improvement from its original source is that the ChaCha quarter-round diffuses changes through bits more quickly than the Salsa20 quarter-round. [3]. The ChaCha quarter round has the same number of adds, xors, and bit rotates as the Salsa20 quarter-round, but the fact that two of the rotates are multiples of 8 allows for a small optimization on some architectures including x86 [27]. ChaCha20 does not provide authentication; for this reason, ChaCha20-Poly1305 was created.

Poly1305 is a universal hash family designed by Daniel J. Bernstein in 2002 and can be used as a one-time message authentication code to authenticate a single message using a secret key shared between sender and recipient. It was first used in conjunction with AES-128 [2] but it was used later on with ChaCha. It was used along with ChaCha20 to create ChaCha20-Poly1305, which is an authenticated encryption algorithm with associated data (AEAD) [12, 30].

2.6.3 AES. The Advanced Encryption Standard (AES), originally known as Rijndael, is a symmetric encryption specification adopted by the U.S. National Institute of Standards and Technology (NIST) in 2001 for securing electronic data [10].

AES is included in the ISO/IEC 18033-3 standard and became effective as a U.S. federal government standard in 2002.

AES is not considered a lightweight cipher by modern standards, but many microcontrollers offer hardware acceleration that significantly improves its performance for these embedded systems [15].

AES has two main modes of operations, AES-CBC (cipher block chaining) and AES-GCM (Galois/Counter Mode). AES-CBC encrypts data in blocks and chains them together, each block depends on the previous one. This mode does not provide authentication and is mostly seen in older systems. AES-GCM, on the other hand, does provide authentication and confidentiality, this mode has become the modern standard for AES encryption.

2.7 Microcontrollers and Resource Constraints

A microcontroller (MCU) contains all the necessary components to create a complete computer system, it is often referred to as a "computer on a chip" [11]. A microcontroller is considered a resource-constrained device because it has limited hardware resources compared to conventional computers. These hardware limitations affect the amount of power, memory, and processing that these devices can use.

Most MCUs have a CPU clock speed of around 200MHz, which is a fraction of the power conventional computers can output. For comparison, a laptop bought in the current year (2025) could have a CPU clock speed of 2.5GHz and contain multiple cores. MCUs also have very limited RAM memory (2KB - 512KB), usually laptops and desktop computers have anywhere between 8 GB and 32 GB of RAM memory. These constrained devices can usually run small programs with no issues, but cryptographic algorithms are very resource intensive and require large amounts of processing power and memory which these devices cannot spare.

Most IoT devices, which are a cluster of resources-constrained components, sensors, and machines [6], run on microcontrollers as they do not consume much power and can perform most required tasks.

3 THREAT MODEL

This report proposes a realistic threat model for a peer-to-peer file sharing program for microcontrollers connected over a WiFi network. To design an effective and secure system. This section shares the extent of the attacker's capabilities within the network.

3.1 Attacker Goals and Capabilities

The main assumption is that the attacker seeks to compromise the confidentiality of peers on the network. To achieve this goal, the attacker may be able to perform the following actions:

- Eavesdrop on communications to read the contents of transmitted files or messages.
- Modify captured messages.
- Replay previously captured messages to spoof legitimate communication.
- Impersonate clients using their IP address.

It is assumed that the attacker will know the IP address of each peer, along with the port used for communication, which is standard in many threat models where the network infrastructure is not hidden from malicous parties.

Furthermore, an attacker may look to compromise the availability of the network; therefore, an attacker may be able to perform the following action:

- Send a malicious payload to a peer's server that exploits a code vulnerability, such as a buffer overflow, resulting in remote code execution or a crash.

With these capabilities, the attacker is able to perform the following attacks:

1) Man-in-the-middle (MitM) attack
2) Replay Attacks
3) IP Spoofing/Impersonation
4) Memory Exploits (i.e., buffer overflow)

3.2 Out of Scope Attacks

It is assumed that the hardware in use does not have any known vulnerabilities or exploits. The WiFi network is also assumed to be robust and has strong security, such as WPA3 [18]. Attacks that produce a denial of service due to an excessive amount of requests being made to the server or client are also out of scope. The reasoning for this is that an attacker could simply drop every packet being sent by the server or client. Therefore, the following attacks are not considered for this project:

- Physical attacks on the hardware components that belong to the system, such as routers or microcontrollers.
- Attacks on the WiFi infrastructure, such as password spraying or other types of attacks that may affect the availability of the WiFi network.
- Denial of service attacks that are produced by sending excessive amounts of data to one of the peers.

Although the project code is open source and includes sample keys for communication between peers, it is assumed that in real-world use, these keys will be replaced and kept secret from potential attackers.

4 PROGRAM OVERVIEW

The system is a peer-to-peer file sharing application which was designed to run on an ESP32 microcontroller [9], this microcontroller model was chosen as it is one of the most popular microcontrollers and its wide range of compatible peripherals such as SD card modules. The application, which was originally designed and tested for two peers, was also designed to be easily modifiable to handle a larger amount of connections.

The program has three different versions; these versions are almost identical to each other; the difference lies on the cryptography libraries being used. The rest of the program structure has been kept consistent to ensure fairness when running tests. The different encryption and hash functions that were used for each implementation are available in table 1

Table 1: ***Comparison of the three different program implementations.***

Impl.	*Payload*	*Encryption Scheme*	*Hash Function*
AES Config	Message/Hash	AES-GCM	SHA256
	File	AES-CBC	
ASCON Config	Message/Hash	ASCON	ASCON-Hash
	File	ASCON	
ChaCha20 Config	Message/Hash	ChaCha20-Poly1305	SHA256
	File	ChaCha20	

The high level overview of the program is as follows:

- The microcontroller connects to a WiFi network and initializes a server listening on port 5000.
- The user is prompted to input the IP address of a remote peer, and a shared encryption key is derived.
- A TCP connection is established between the local client and the remote peer's server.
- The client sends encrypted commands, such as file requests, using the shared key and a nonce to prevent replay attacks.
- The server responds by computing and sending an encrypted hash of the requested file for integrity verification.
- The server encrypts and sends the file in chunks; the client writes these chunks into a new file then proceeds to decrypt it and verifies the file's integrity before storing it indefinitely.

4.1 Initial Connection

The program starts its execution by first connecting to a WiFi network. After a successful connection, the microcontroller's assigned IP address will be printed to the terminal. Following this, the server will start on port 5000 and listen for any incoming connections. At this time, the user will be prompted for the IP address of the remote server to which they want to connect to. The shared key is

also computed and will be used to encrypt all messages and files during this connection. Figure 3 shows this process on a flowchart diagram.

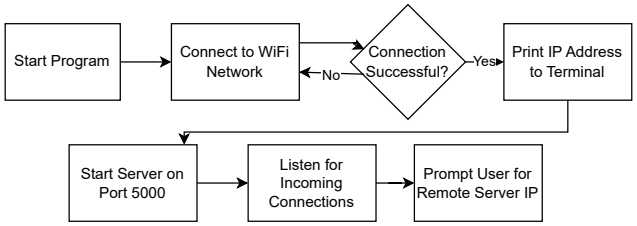


Figure 3: ***Flowchart of initial program execution.***

4.2 Sending a File Request

Once a client connects to a peer's server, they will establish a TCP connection which will be used throughout the program's lifetime. At this stage, the user can begin communicating with the server. The following command can be issued to download a file from the remote server.

`/GET filename.extension`

This message is encrypted using the shared key and a nonce, the encrypted message is subsequently sent over the network to the server. Once the server receives the message it will decrypt it and verify that the nonce has not been seen previously. Figure 4 shows this process.

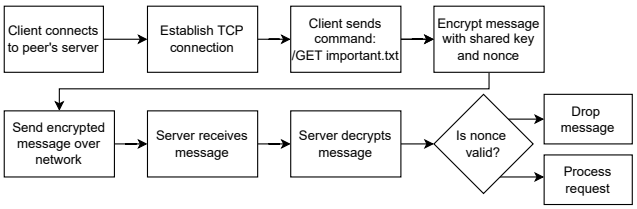


Figure 4: ***Process of sending a file request.***

4.3 Sending the File Hash and the Encrypted File Contents

The server will proceed to verify that the file being requested exists in its file system and will begin to compute the hash value of the file. This hash is then encrypted and sent to the client. The client can then use this hash to verify that the file that was downloaded is the same as the one the server sent over the network. This process can be seen in figure 5.

Before sending the file contents over the network, the server first starts encrypting the file. Once this step is completed, it will send chunks of the encrypted file to the client until all the file has been sent. The client will receive these chunks and append them to a file in its own file system. Once the last chunk has been received, the client will begin decrypting its contents using the shared key, and it will save the decrypted file to the SD card. The client will compute the hash of the decrypted file and compare it to the hash

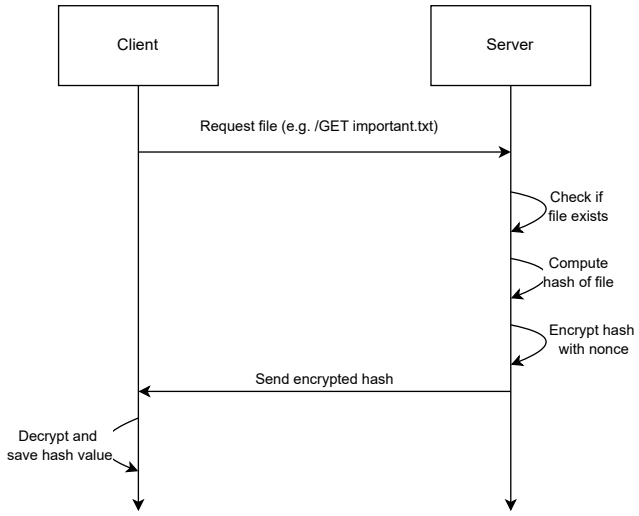


Figure 5: ***Process of sending a file hash.***

that was sent previously by the server. If the hashes do not match, the file will be deleted from the file system. Figure 6 shows a UML diagram that describes this implementation.

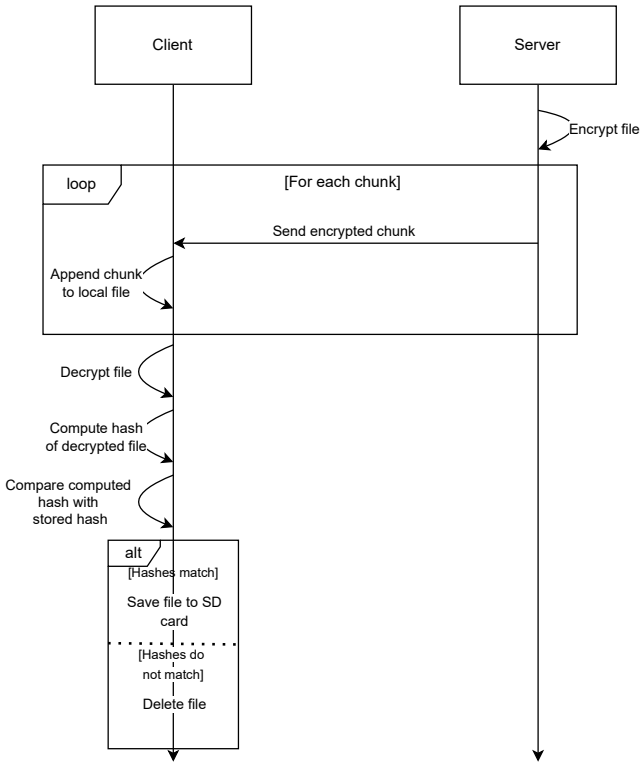


Figure 6: ***UML diagram showing the process of sending a file.***

5 SECURE PROTOCOL DESIGN AND SYSTEM IMPLEMENTATION

5.1 Arduino IDE and Serial Terminal

The program was developed using Arduino IDE, which simplifies the process of compiling code and transferring the compiled binaries to the microcontroller. This is achieved using the xtensa-esp32-elf-gcc compiler and a Python script, esptool.py, which transfers the binaries to the microcontroller via a USB connection. Once the binaries are written to the flash memory of the microcontroller, they can be executed.

The serial terminal enables communication between the user and the microcontroller. Program output can be printed to the screen, allowing the user to receive information. The terminal is also used to accept user input, which can be used to execute commands or modify variables within the program's context.

5.2 Microcontroller Choice

For this project, the selected microcontroller was the ESP32 [9], more specifically the DEVKIT DOIT model, which offered the WiFi capabilities needed to communicate over a WiFi network. This MCU has a dual core CPU that operates at 240MHz and a RAM memory capacity of 512KiB, which introduced challenges when dynamically allocating memory during the program's execution. To store files, a simple 64GB microSD card was used on each MCU, which ensured sufficient capacity for most files. The microSD cards were able to be connected using the microSD card module, made possible due to the MCU being able to connect with a wide range of peripheral modules.

5.3 TCP connections

While libraries like MbedTLS and WolfSSL provide full SSL/TLS implementations for secure client-server communication, this project focuses specifically on evaluating their cryptographic primitives. Therefore, only their encryption, decryption, and hashing functions were used to build a simplified, custom TLS-like protocol over a standard TCP connections.

5.4 Message metadata and structure

When sending an encrypted message, hash or file chunk some metadata is appended to the message. The metadata fields change depending on the payload being sent. The metadata and structure of the payload were created exclusively for this project and were not taken from another already existing system.

Figure 7: *Payload structure for a get request or hash.*

Figure 7 shows the payload structure of a GET request. In the Ascon implementation, the nonce has a size of 16 bytes, while for the Mbed TLS and WolfSSL implementations, the nonce is 12 bytes. When the client receives a message, it first reads the initial 4 bytes, which indicate the total length of the payload. This approach is beneficial because TCP is a stream-oriented protocol, and messages can arrive in segments. By knowing the total payload size in advance, the client can continue reading from the socket until the full message is received, ensuring that the entire encrypted payload is correctly reconstructed before decryption.

The message type field helps the client or server distinguish between different types of encrypted payloads. Based on the numeric code received, the system determines how to process the message appropriately.

Messages and hashes are encrypted using a nonce to prevent an attacker from using replay attacks. The server will keep track of observed nonces and will drop any payload that contains an invalid nonce, furthermore, messages and hashes are encrypted using authenticated encryption, if the message is modified in any way the decryption function will output an error, as the authentication tag will not match and will not decrypt the message. This is true for all three libraries. The metadata fields of these payloads are included inside the associated data so that any tampering of these fields will be detected during the decryption process.

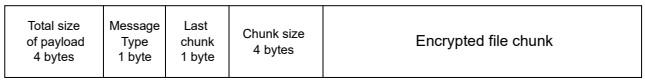


Figure 8: *Payload structure for sending a file chunk.*

Figure 8 shows the payload structure of a file chunk. Similarly to the previous structure, it contains metadata fields such as total size and message type.

File chunks, unlike request messages, do not contain a nonce field. This design decision was made to optimize performance, as generating and managing a unique nonce for each chunk along with the corresponding encryption and decryption overhead would significantly impact the efficiency of file transfers on resource-constrained devices. Given that the encryption is performed using a secure secret key and each chunk is transmitted sequentially within a single session, the absence of per-chunk nonces is considered an acceptable trade-off in this context.

The last chunk field is used to warn the client that the current chunk is the last one it can begin decrypting the file.

Although the chunk size field may appear to be redundant, given that the metadata already contains the total size field, it serves a specific purpose: it explicitly indicates the number of bytes that make up the encrypted file chunk. In contrast, the total size field includes the size of the metadata. Since the size of the metadata fields may change over the development of the application or be updated in the future, keeping a separate chunk size ensures some consistency when parsing the payload.

The encrypted file chunk is kept at a constant size of 512 bytes except for the last chunk, which may be of smaller size. This size was chosen as it is smaller than the maximum segment size of a TCP packet. Changing the chunk size to other values did not greatly affect the transmission speed and is not the main focus of this paper.

5.5 File Encryption & Decryption

File encryption and decryption is performed in chunks; this is due to the microcontrollers not having the memory capacity to read and store all the contents of the file at once.

5.5.1 Ascon Implementation. . In the case of the Ascon implementation, the library already includes built-in support for encrypting and decrypting file contents [36]. This program also encrypts and decrypts the file contents in chunks. A C macro is used to define the chunk size, which varies depending on the target platform; for the microcontrollers used in this work, it resolves to 128 bytes. A minor modification was made to the original source code to allow reading data directly from the SD card.

Ascon uses the incremental version of ASCON-80pq to encrypt files which uses an authenticated encryption scheme. The hash value in this case is not needed but is kept in place as it serves for a nice comparison.

5.5.2 AES & ChaCha20 Implementation. Large files are encrypted and decrypted by processing them in fixed-size chunks. This chunk-based approach continues iteratively until the entire file is processed. A chunk size of 2048 bytes provided the best performance for both cryptography functions. This chunk-based mechanism was designed specifically for this project to accommodate the limited memory available on the microcontroller. The official Ascon reference implementation served as a guideline for structuring this approach, particularly in handling incremental encryption on constrained devices.

The encryption algorithms used to encrypt files do not provide authentication; this decision was made to improve performance and reduce the complexity of the implementation. The main assumption is that authenticated encryption is more resource intensive, because the program uses many encryption and decryption calls for large files, a small improvement quickly adds up. To compensate for the lack of authenticated encryption, file integrity is verified using the hash value the server sends to the client, which is encrypted using an authenticated encryption scheme. This allows the client to verify that the file was not modified during transit and can be kept in storage. These decryption functions, as they do not use authentication, will not output an error as the decryption cannot "fail"; therefore, it is important to verify the integrity of these downloaded files.

5.6 File Hashing

Similar to encrypting files, the program due to the memory constraints of the embedded system, is unable to hash the entire contents of the file in one iteration. The program will again process the files in chunks to produce a hash value.

The AES and ChaCha20 implementations compute the SHA-256 hash of the file, which remains a secure and widely trusted cryptographic hash function by current standards [19]. The AES version uses WolfSSL's SHA256 function whilst the ChaCha20 implementation uses MbedTLS's. This approach avoids mixing libraries unless strictly necessary and also demonstrates how the same hash function can vary in performance depending on the underlying library.

Ascon, on the other hand, uses the 256-bit Ascon-Hash algorithm. In all implementations, the hash computation is performed incrementally by feeding data in chunks into the algorithm. A chunk size of 2048 bytes was chosen, as it provided the best time efficiency for all implementations.

5.7 Key Generation & Management

5.7.1 Key Management. The application was built to use predefined, hardcoded keys which are stored inside the program's code. While this mitigates the risk of a man-in-the-middle attack, it introduces new security risks. If an attacker is able to exploit a buffer overflow or similar memory exploits he or she could leak these keys and be able to reverse engineer the shared secret.

5.7.2 Key Generation. Each microcontroller is assigned a key pair Using the client's private key and the remote server's public key a shared key is computed using Elliptic Curve Diffie-Hellman (ECDH) [17] which is a more efficient version of the original Diffie-Hellman key exchange for embedded systems. All implementations of the program generate this shared secret using mbedTLS's ECDH implementation. This is the only section of the program where another cryptography library was used. There is no key exchange in this case, per se, as the shared key is computed by each peer without needing to send any data to the other as the public key of the other peer is already known. An explanation of ECDH can be found below.

5.8 Elliptic Curve Diffie-Hellman Key Exchange

Let:

- d_A: Alice's private key
- Q_A: Alice's public key
- d_B: Bob's private key
- Q_B: Bob's public key

The shared secret is computed as:

$$\text{Alice computes: } S = d_A \cdot Q_B$$

$$\text{Bob computes: } S = d_B \cdot Q_A$$

Since scalar multiplication is commutative in this context:

$$d_A Q_B = d_B Q_A$$

Both parties derive the same shared secret point S on the elliptic curve.

In the Ascon version, encryption requires a 16-byte key; therefore, only the first 16 bytes of the generated 32-byte shared key are used for this version of the program.

5.9 Nonces

Nonces are utilized to prevent replay attacks by ensuring that each encrypted message is unique. While these encryption function use 12-16 bytes for nonces, the application takes a conservative approach, given the assumption that no client will send more than 2000 messages per session, the server tracks up to 2000 unique nonces per client connection. To simplify this, only the last 4 bytes of the nonce are used to ensure uniqueness, reducing the nonce space from 2^{128} to 2^{32}, of which only 2000 values are actively used.

6 EVALUATION

The experiments were performed utilizing two microcontrollers, which would concurrently run the same version of the file sharing program (i.e., Ascon, AES, ChaCha20). Depending on the experiment, the microcontroller would need to perform tasks such as encryption, decryption, and hashing of different files.

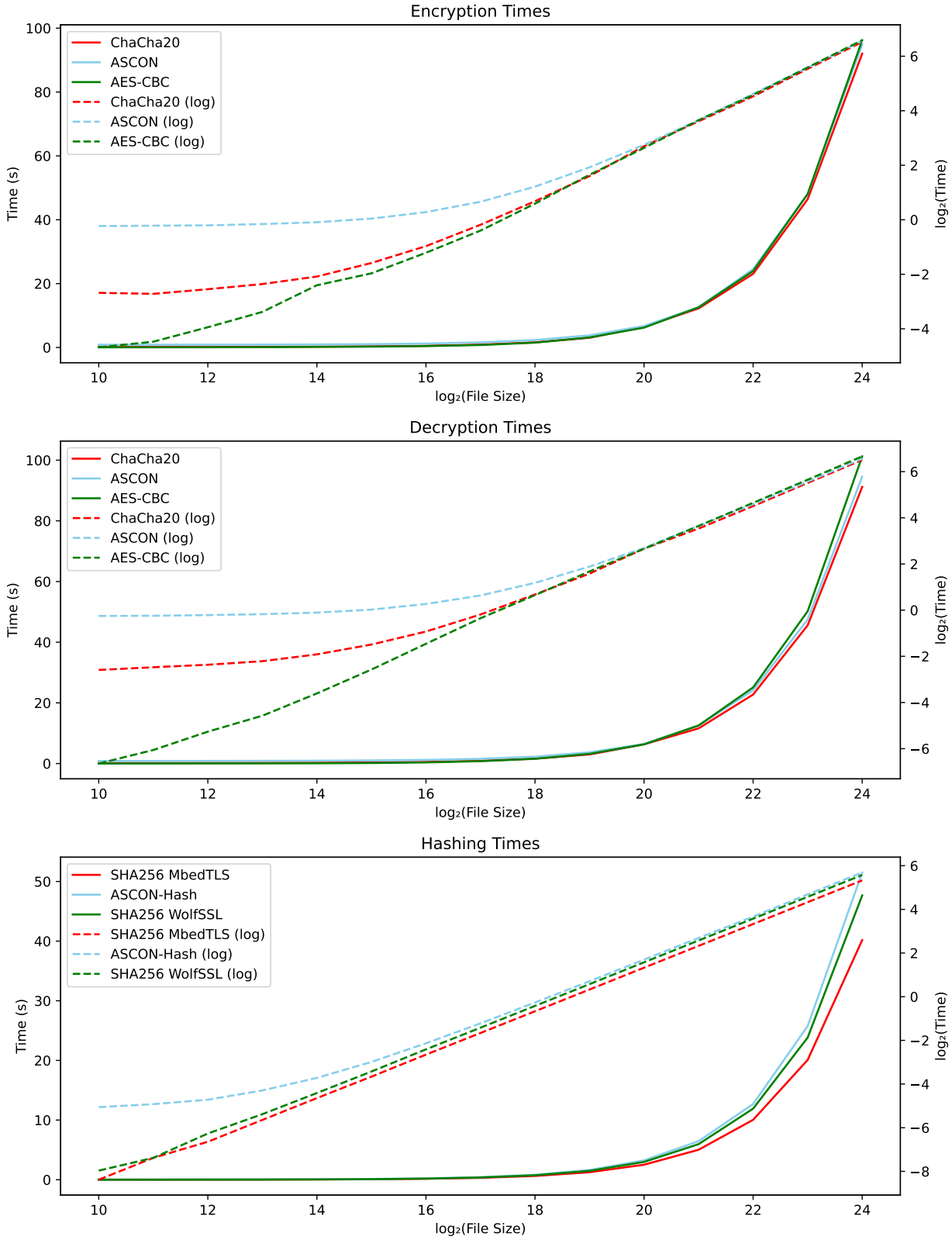


Figure 9: Comparison of each implementation for encryption, decryption, and hashing for files of different sizes.

6.1 Time Efficiency Comparison

To test the efficiency of these libraries, files of different sizes ranging from 2^{10} bytes to 2^{24} bytes were generated, which contained random data inside them. This range was chosen to assess performance across varying use cases, from very small payloads to large files that require significant time to encrypt or decrypt. It highlights how each library behaves under both lightweight and computationally intensive conditions. To record the time needed to complete each operation, the built-in `millis()` and `micros()` functions for Arduino boards were used, which record the time in milliseconds and microseconds, respectively. The experiment was repeated five times and the results were averaged.

Figure 9 shows the time taken by each implementation to complete the core cryptographic functions. A logarithmic plot is also included to highlight how the execution time of each library scales with increasing file size.

6.1.1 Encryption & Decryption. It can be seen from the first graph in figure 9 that all encryption and decryption functions performed similarly to each other, especially as the file size increases, the functions seem to converge with minimal deviations. It can be observed from table 3 that encrypting or decrypting the largest file results in similar times for all functions with the largest difference being 4.6% for encryption and 11.1% for decryption, with ChaCha20 being the fastest and AES-CBC the slowest. However, for smaller file sizes, performance differences become more pronounced. Ascon exhibits the worst performance as it seems to have a significant overhead, this can be visualized by looking at the logarithmic plot in figure 9 and observing the flat line for ASCON that spans from file sizes of 2^{10} bytes to 2^{16} bytes. Table 2 shows that Ascon is roughly 22 and 84 times slower compared to AES-CBC for encryption and decryption, respectively. AES-CBC performs best when it comes to smaller file sizes.

Table 2: Execution times (in seconds) for the smallest file size (2^{10} bytes)

Implementation	Encryption	Decryption	Hashing
ChaCha20 config	0.156	0.166	0.003
ASCON config	0.852	0.840	0.030
AES config	0.039	0.010	0.004

It is also worth noting that AES-CBC is slower at decryption than encryption, which is not the case for ChaCha20 and Ascon. This was to be expected, as the hardware manual for the ESP32 states the following: "The AES Accelerator requires 11 to 15 clock cycles to encrypt a message block and 21 or 22 clock cycles to decrypt a message block" [15]. This suggests that decryption could be between 40% and 100% slower than encryption. However, in practice, for the largest file size specifically, only a 5.2% increase in decryption time was observed.

6.1.2 Hashing. The last graph from Figure 9 shows the hashing times for all implementations. SHA256 is faster for all file sizes, with MbedTLS's implementation being the fastest overall. Similarly to encryption and decryption, Ascon seems to have a time overhead, although smaller, it still slows down the hash computation for smaller file sizes. The logarithmic plot shows that the SHA256 functions follow a straight line which indicates a linear relationship, Ascon, on the the other hand, only starts having a linear relationship after a file size of 2^{17}. In general, hashing is a more time-efficient function, as can be seen in tables 2 and 3.

Table 3: Execution times (in seconds) for the largest file size (2^{24} bytes)

Implementation	Encryption	Decryption	Hashing
ChaCha20 config	92.036	91.185	40.168
ASCON config	94.798	94.594	51.497
AES config	96.239	101.259	47.655

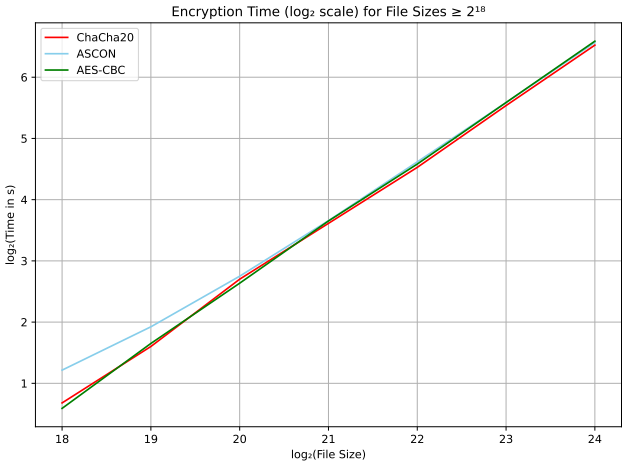


Figure 10: *Encryption times for file size greater than* 2^{18} *bytes.*

6.2 Linear Relationship

We can quickly prove that these functions scale linearly for larger file sizes by viewing that the logarithmic plot becomes a straight line as the file size increases, as can be seen in Figure 10. This graph shows the times for encryption after the logarithmic line becomes a straight line, which is then used to find the gradient of the line and the Y intercept. We can compute it as follows: Given the two points:

$$(21, 3.613), \quad (24, 6.524)$$

Assuming a linear model of the form:

$$y = mx + c$$

We substitute the values into the equation:

$$3.613 = 21m + c \tag{1}$$

$$6.524 = 24m + c \tag{2}$$

Subtracting equation (1) from equation (2):

$$6.524 - 3.613 = (24m + c) - (21m + c)$$
$$2.911 = 3m$$
$$m = \frac{2.911}{3} = 0.9703$$

Substitute $m = 0.9703$ into equation (1):

$$3.613 = 21 \cdot 0.9703 + c$$
$$3.613 = 20.3763 + c$$
$$c = 3.613 - 20.3763 = -16.7633$$

Final result:

$$m \approx 0.9703, \quad c \approx -16.763$$

The slope $m \approx 0.9703$ comes from a log-log regression. Since this computation was done in logarithmic space, it implies a power-law relationship in normal, linear space.

$$t \propto \text{size}^m$$

Therefore, with $m \approx 0.9703$, we have:

$$t \propto \text{size}^{0.9703}$$

This shows that the time grows almost linearly with the file size for encryption Since the exponent is slightly less than 1, the growth is sublinear, which means that it is slightly more efficient than linear growth.

While the computed y-intercept $c \approx -16.763$ may appear large, it is important to note that is it in a logarithmic y-axis, this value corresponds to approximately $2^{-16.763} \approx 9.0 \times 10^{-6}$. This suggests that the y-value (the time) at a file size of 1 byte is nearly 0, indicating negligible time for extremely small inputs, which makes sense given the linear relationship. This maybe true for AES-CBC as it's logarithmic line does follow a more straight line for smaller files. However, the same does not apply to Ascon and ChaCha20, whose plots flatten at smaller file sizes, indicating that the computed y-intercept does not accurately reflect their behavior in that region.

Although the computation of the gradient and the y-intercept for decryption and hashing is not shown, results also show the same behavior as the encryption function shown earlier. This shows that all these functions scale linearly.

6.3 CPU Usage Test

To record the CPU utilization metric, a function running on a separate core of the microcontroller was used. This function was executed in a loop, printing CPU usage information every half a second, with the output representing CPU utilization in percentage points.

The CPU usage rapidly increases, as observed in (Fig. 11). This was tested on all functions and libraries, but always yielded the same graph. This was expected as these cryptographic functions are computationally intensive and require a lot of resources. For large files, these functions execute for long enough that the CPU reaches almost 100% usage.

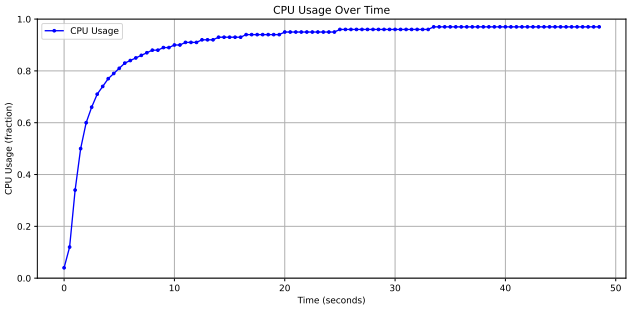


Figure 11: ***CPU usage over time during a resource intensive operation.***

6.4 Memory Usage

Memory usage was measured by recording the amount of free memory available before executing the function under test. These implementations were tasked with encrypting, decrypting, and hashing a file of size 2^{22} bytes. This measurement was repeated in a loop during the function's execution, and the lowest observed amount of free memory was taken as the value for comparison. It is worth noting that some libraries may allocate a large buffer prior to execution to avoid frequent memory requests from the MCU. As a result, they may not use all the allocated memory, which can increase their memory usage.

Table 4: ***Memory usage (in bytes) before and during cryptographic operations for each library. Usage is computed as the difference between memory before and during.***

ChaCha20 Config

Operation	*Before*	*During*	*Usage*
Hash	207900	200740	7160
Encrypting	207868	197616	10252
Decrypting	207868	197616	10252

AES Config

Operation	*Before*	*During*	*Usage*
Hash	207892	200608	7284
Encrypting	207868	197724	10144
Decrypting	207868	197724	10144

Ascon Config

Operation	*Before*	*During*	*Usage*
Hash	207900	200908	6992
Encrypting	206876	196732	10144
Decrypting	206876	196732	10144

Comparing these lightweight cryptography libraries to a more standard one like OpenSSL would have been informative. Unfortunately, importing the OpenSSL library to the microcontroller was not feasible, as it currently does not provide support for these embedded systems. As a result, the memory usage test was performed

on a standard laptop by encrypting a file of size 2^{22} bytes. Simple experiments show that OpenSSL uses more than 20 times more memory than the lightweight libraries tested. OpenSSL consistently used around 6MB of memory for both encryption and decryption and approximately 5.8MB for generating the SHA-256 hash of the same file.

Table 4 shows the amount of memory each of these implementations uses in bytes. It is clear that hashing uses the least amount of memory, with Ascon using the least amount with 6992 bytes. Encryption and decryption use the same amount of memory which could be caused by the internal function implementations allocating a fixed-sized buffer. In this case, both AES and Ascon allocate the same exact number of bytes and use the least for both encryption and decryption. In general, these libraries use very similar amounts of memory.

6.5 Chunk Size Results

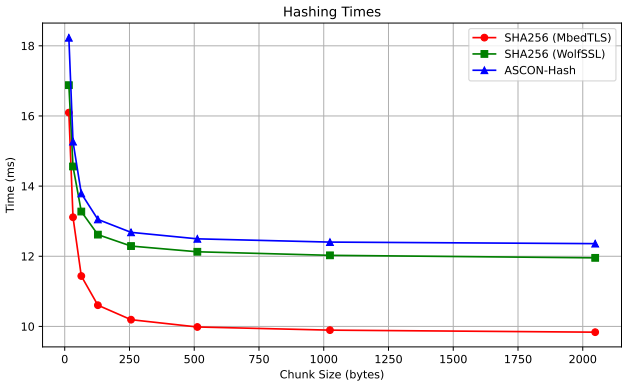
To evaluate which chunk size, which is fed into each function, was optimal for each implementation, each configuration was tasked with encrypting, decrypting, and hashing a file with size of 2^{22} bytes. Note that chunk size in this context means the amount of bytes that is passed to the corresponding function, which is just a section of the file that is being processed, these functions may internally split this string of bytes further into smaller chunks.

Since Ascon already has their own implementation for file encryption and decryption, it was assumed that the chunk size this library uses is already optimized, therefore, only chunk size for hashing was tested for this library, Chunk sizes ranging from 16 bytes to 2048 bytes were tested. Chunk sizes larger than 2048 bytes could sometimes crash the program as it was unable to allocate the necessary amount of heap memory for that chunk, therefore, chunks could only have a maximum size of 2048 bytes which allowed the program to execute normally without any crashes.

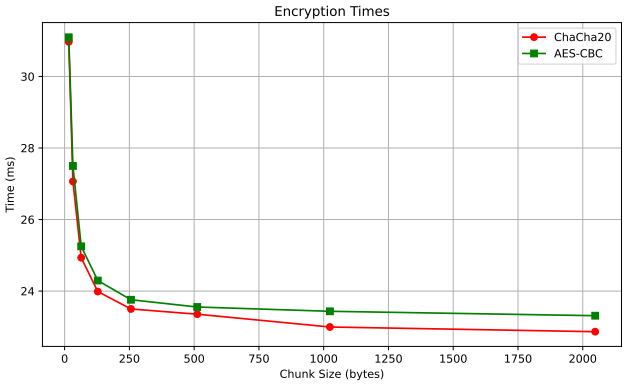
Figure 12 shows three graphs illustrating the performance of each implementation as a function of chunk size. Specifically for hashing, a larger chunk size is optimal, leading to a time improvement of 32.22%, 29.2% and 38.9% for Ascon, SHA256 (WolfSSL implementation), and SHA256 (MbedTLS implementation), respectively. It can also be seen that the improvement rate decreases as the chunk size increases. It is interesting to see that even though MbedTLS and WolfSSL are using the same function, the overall improvement is different.

We can see a similar performance gain with ChaCha20's encryption and decryption functions, which also benefited from a larger chunk size. Both functions benefited equally by 26.2%.

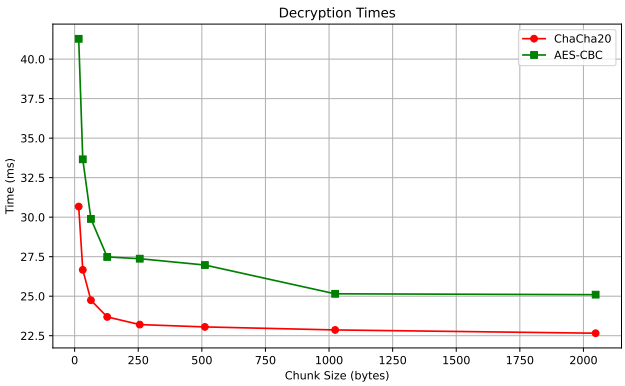
AES-CBC saw great improvements when increasing the chunk size of 25% and 39% during encryption and decryption respectively; similar to the other implementations, it improves rapidly with a small increase in the chunk size until the rate of improvement stalls at around 1024 bytes. It is interesting that decryption benefited a lot more than encryption, almost twice as much. If instead a 16-byte chunk was chosen, decryption would be 33% slower than encryption, which is closer to the range the ESP32 hardware manual suggests for AES acceleration (40% - 100%), this estimate is based on the clock cycles needed to complete each operation. [15].



(a) Hashing times



(b) Encryption times



(c) Decryption times

Figure 12: Execution times of hashing, encryption, and decryption for various chunks sizes.

Originally, it was thought that using large chunks would not help improve performance as allocating such large blocks of memory would take a long time, but this was not the case. Table 5 has data showing the time taken in microseconds to allocate different memory sizes. We can see that allocating these small amounts of memory, even for an embedded system, is still incredibly fast, even

when allocating larger amounts of memory, it is able to do so in constant time. Following this, it was thought that writing memory on this newly allocated memory region could be a bottleneck, but it was also not the case; the same table shows that while the time taken increases linearly, it is still negligible.

As previously discussed, larger chunks contribute to a better performance in all three configurations, consequently, a chunk size of 2048 was chosen for all implementations.

Table 5: *Time taken in (µs) for* `malloc()` *and* `memset()` *at various allocation sizes.*

Allocation Size (bytes)	`malloc()`	`memset()`
16	6	0
32	6	0
64	6	0
128	6	0
256	6	1
512	6	1
1024	6	2
2048	6	4

6.6 Security Properties

6.6.1 Man-in-the-middle attack. An attacker can read the data being sent from one client to another. This is a threat to confidentiality as any message or file being sent over the network without encryption could be read by an attacker which could expose sensitive information. To prevent this, files and messages use encryption to avoid anyone without the shared key from decrypting the data. This is achieved in each implementation by integrating the corresponding cryptographic scheme, Ascon, ChaCha20, or AES into the same core program structure.

In the case that an attacker modifies a message or hash value, the receiver will notice as these payloads use authenticated encryption with associated data (AEAD) meaning that the decryption will trigger an error. If the attacker modifies the contents of a file that is in transit the client will decrypt it but the hash value of the decrypted file will not match with the one the server sent originally, meaning that the files are not the same.

6.6.2 Replay Attacks. Since the attacker does not have access to the shared key to encrypt messages, the attacker is unable to craft a payload that will trigger the server to process the attacker's request, therefore, an attacker could resort to a replay attack, by analysing the requests being made it can be determined which payload triggers the server to start sending files or performing some other resource intensive task. If an attacker copies and sends the same request the server would treat it as a valid request and would drain valuable resources from the server. To prevent this, nonces are used to encrypt messages. If a nonce has already been seen the message will be dropped and will not be processed any further. Nonces also make sure that even if the same plaintext is encrypted, the computed ciphertext is different.

6.6.3 IP Spoofing/Impersonation. An attacker could start sending requests with by impersonating a valid client IP address, but this will not work as the attacker is unable to encrypt messages or hashes with the correct cryptographic key, therefore, any message sent by the attacker will trigger an error during decryption. If an attacker sends a malformed file chunk, the client will detect the unusual chunk and simply ignore it as the hashes do not match.

6.6.4 Memory Exploits. The program, overall uses safe coding practices as much as possible. Some payloads like hashes and files have a certain maximum length, any hash for file whose length is above it is discarded, and messages after a certain length are assumed to be malicious and are therefore dropped.

Memory that is no longer in use is freed to reduce memory usage and prevent security issues such as use-after-free or memory leaks. Proper deallocation helps minimize the risk of dangling pointers and other memory-related vulnerabilities.

7 DISCUSSION

7.1 Metadata

The metadata section of an encrypted chunk is inefficient. Instead of using an entire byte of data to signal if the current chunk is the last one of the encrypted file, it could instead use a bit. This would save 7 bits of data per chunk. While this mainly affects the connection performance it still impacts the MCU directly when processing the metadata.

7.2 Keys

In the initial implementation, peers dynamically generated public-private key pairs at runtime. Upon establishing a connection, the server and client would exchange their public keys and compute a shared secret key using a key exchange protocol. However, this design is inherently vulnerable to man-in-the-middle (MITM) attacks. An attacker positioned between the peers could intercept and replace public keys, allowing them to derive the shared key and decrypt the communication.

The standard solution against such attacks involves authenticating public keys through certificates issued by a trusted certificate authority (CA). Unfortunately, due to time constraints, integration challenges, and the complexity of implementing certificate-based authentication on the resource-constrained MCU, this feature could not be implemented in the project.

Instead of relying on hardcoded keys, using digital certificates would offer a more secure and scalable solution, enabling peers to verify the authenticity of each other's public keys before computing the session key [21].

Keys are not stored on the file system to mitigate the risk of file disclosure vulnerabilities. If an attacker can exploit a vulnerability in the server, such as a file disclosure, they could potentially download arbitrary files, including sensitive ones like the server's private key. This would allow the attacker to derive the shared key and decrypt all data sent by the server or client and impersonate the server. According to Kerckhoffs' Principle, the security of a cryptosystem must lie solely in the secrecy of its keys; all other aspects, including the algorithm itself, should be considered public knowledge [29].

Since the keys are stored on the code and therefore on the RAM of the microcontroller, if a buffer overflow occurs, an attacker may be able to read the region of memory that stores the private keys

of each client, which would allow the attacker to decrypt all the messages and impersonate other peers.

7.3 WiFi Network

The microcontrollers that were used to test the implementation of the file sharing program (ESP32) can only connect to WiFi networks that operate at the 2.4 GHz range. Since most home networks operate at the 5 GHz frequency it made it impossible to use this robust and fast network, some routers do allow the owner to split the network into two and gain access to the 2.4 GHz option, but this was not possible for this project. Therefore, a mobile hotspot had to be used to connect the microcontrollers, which did have an option to operate at 2.4 GHz. This proved difficult as this mobile hotspot is less reliable and slower than a conventional router. This does not affect the cryptographic functions as they are performed offline, but it did affect the TCP connection that was used to communicate between the microcontrollers, sometimes causing the peers to disconnect, a more reliable network should be chosen in the future.

8 RELATED WORK

[28] shows an implementation of an embedded client-server system for home applications using the file transfer protocol (FTP), SNTP and HTTP. The system described in [28] operates using a 32 bit RISC microprocessor, which is an embedded system similar to the ESP32 microcontroller that was used for this project.

[16] also cover a lightweight FTP implementation which is suitable for resource-constrained devices, it also covers the architecture of the lightweight file transfer protocol implementation as well as the encryption aspect of this system.

[23] shows the importance of encryption on embedded systems, showing different encryption and hashing functions such as AES, MD5 and BlowFish. This work also shows the encryption performance when encryption files of large sizes and different password lengths are encrypted.

Bhuvanagiri Udayakumar [7] evaluates the use of a variant of ChaCha20, specifically XChaCha20, for encrypting file systems and large files for cloud services. The work in [7] also compares it with different encryption algorithms, one of them being AES's software version, in this report used AES's hardware version as the program was built for embedded systems, otherwise, AES would have been much slower. The paper found that XChaCha20 was more efficient and secure than AES to encrypt large amounts of data. This project also found that ChaCha20 was faster than AES although the differences were smaller it can be attributed to the different systems were the tests were conducted, using different variations of function, and using hardware accelerated AES.

The work of Anil Wurity and L. Sumalatha in [40] shows how Ascon performs when encrypting and decrypting files. Their work found that for a file of a size of 1 MB, Ascon takes 9 seconds and 9.5 seconds to encrypt and decrypt, respectively. The experiments carried out in this project show that there was no significant difference between the times taken for encryption and decryption for Ascon; another difference is that for a file of the same size, the experiments performed in this project with a file of the same size show a time of 6.7 seconds for both encryption and decryption instead.

9 CONCLUSION

This thesis presented the design and implementation of a secure file sharing program for a network of embedded devices. The primary objective was to create a secure and reliable program to share files between devices even when a malicious party has many capabilities to attack the system, such as listening on all communications on the network, replaying messages, modifying captured messages, etc. As a secondary objective, this thesis also shows how different implementations of the program perform when executing different cryptography operations such as encryption, decryption, and hashing by comparing different encryption schemes like AES, Ascon, and ChaCha20. Parameters such as chunk size, memory usage, CPU usage, and time efficiency were tested.

The system demonstrates how the discussed attacks were taken into account when designing the system and how they are prevented by using encryption and hashing to prevent an attacker from compromising the availability and confidentiality of the peers in the network. These attacks are prevented by using authenticated encryption for messages which protects the integrity and confidentiality of the data inside of them. In the case of files confidentiality if protected by using encryption, integrity is not protected directly but is verified after the file has been downloaded by comparing the hash value of the file to the server's. Further research is needed to confirm the underlying assumption of this system; that authenticated encryption is more time-consuming than non-authenticated encryption.

Experiments showed that all of the different encryption schemes performed to a very similar level for encryption and decryption, with AES-CBC being better for smaller sized files and ChaCha20 being a better option as the file size increases. Ascon also showed great performance, however, the implementation that was tested was slower for smaller-sized files, where it was outperformed by the other implementations. For hashing, the SHA256 implementation from both MbedTLS and WolfSSL showed a more linear relationship throughout the test set. MbedTLS's implementation was faster overall for all file sizes with Ascon-Hash performing the worst out the compared functions.

The system has some limitations that expose it to some attacks. Hardcoded keys can be leaked by a memory leak. Without the use of certificates, the program must trust the public keys provided by the user. It also does not allow the program to scale well as more and more peers join the network.

9.1 Recommendations and Future Work

The results show that encryption and decryption for regular sized files below 1 MB are quite fast and efficient. For smaller files, it is recommended to use cryptography for embedded systems. Larger file sizes are quite slow; therefore, it is only recommended in cases where security has to be strictly maintained or where time inefficiency is not an important requirement.

Chunk sizes larger than 2048 bytes are not recommended as they proved to be unstable; on the other hand, if the microcontroller on which the system is being run has a larger RAM capacity, it would be recommended to increase the chunk size as the time efficiency increases. More research is needed to determine the ideal chunk size

that ensures both system stability (i.e., no crashes) and maximum time efficiency.

The metadata structure, as explained before, is not refined, although the file transfer efficiency and speed were not the main focus of this project, it is still recommended to find a better and more efficient metadata structure for future implementations of the system.

Future work could include testing the same implementation on other embedded devices, implementing certificate based authentication, expanding the current system to allow for more than two peers as well as including more parameters for testing purposes, such as; power consumption and CPU temperature and implementing a more robust nonce tracking system that uses the full byte length which would provide further safety against replay attacks and allow clients to send more messages without running out of available nonces.

REFERENCES

[1] Itay Berman, Akshay Degwekar, Ron D. Rothblum, and Prashant Nalini Vasudevan. 2018. Multi-Collision Resistant Hash Functions and Their Applications. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 133–161.

[2] Daniel J Bernstein. 2005. The Poly1305-AES message-authentication code. In *International workshop on fast software encryption*. Springer, 32–49.

[3] Daniel J Bernstein et al. 2008. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, Vol. 8. Citeseer, 3–5.

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2007. Sponge functions. In *ECRYPT hash workshop*, Vol. 2007.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2012. Permutation-based encryption, authentication and authenticated encryption. *Directions in Authenticated Ciphers* (2012), 159–170.

[6] Isha Bhardwaj, Ajay Kumar, and Manu Bansal. 2017. A review on lightweight cryptography algorithms for data security and authentication in IoTs. In *2017 4th International Conference on Signal Processing, Computing and Control (ISPCC)*. 504–509. https://doi.org/10.1109/ISPCC.2017.8269731

[7] Kahol Gaurav Bhuvanagiri Udayakumar. 2021. *Evaluation of XChaCha20-Poly1305 for Improved File System Level Encryption in the Cloud*. Ph.D. Dissertation. Dublin, National College of Ireland.

[8] William J Buchanan, Shancang Li, and Rameez Asif. 2017. Lightweight cryptography methods. *Journal of Cyber Security Technology* 1, 3-4 (2017), 187–201.

[9] Neil Cameron. 2023. *ESP32 Microcontroller*. Apress, Berkeley, CA, 1–54. https://doi.org/10.1007/978-1-4842-9376-8_1

[10] Joan Daemen and Vincent Rijmen. 1999. AES proposal: Rijndael. (1999).

[11] John H Davies. 2008. *MSP430 microcontroller basics*. Elsevier.

[12] Fabrizio De Santis, Andreas Schauer, and Georg Sigl. 2017. ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications. In *Design, automation & test in europe conference & exhibition (DATE), 2017*. IEEE, 692–697.

[13] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. 2021. Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *Journal of Cryptology* 34, 3 (2021), 33. https://doi.org/10.1007/s00145-021-09398-9

[14] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. 2014. Ascon: Submission to the CAESAR Competition. http://ascon.iaik.tugraz.at. Accessed: 2025-05-31.

[15] Espressif Systems. 2023. *ESP32 Technical Reference Manual*. https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf Accessed: 2025-06-06. Refer to Chapter 14, page 280.

[16] Min Gong, Wei Dong, and Zhen Ya Zhang. 2014. An embedded FTP server: research and implementation. *Applied Mechanics and Materials* 543 (2014), 1977–1980.

[17] Tarun Kumar Goyal and Vineet Sahula. 2016. Lightweight security algorithm for low power IoT devices. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 1725–1729. https://doi.org/10.1109/ICACCI.2016.7732296

[18] Asmaa Halbouni, Lee-Yeng Ong, and Meng-Chew Leow. 2023. Wireless Security Protocols WPA3: A Systematic Literature Review. *IEEE Access* 11 (2023), 112438–112450. https://doi.org/10.1109/ACCESS.2023.3322931

[19] H Handschub and Henri Gilbert. 2002. Evaluation report security level of cryptography–sha-256. *Journal of Women s Health* (2002).

[20] IJEDR. 2014. Cryptography Algorithms: A Review. *International Journal of Engineering Development and Research (IJEDR)* 2, 2 (June 2014), 1667–1672. https://rjwave.org/IJEDR/papers/IJEDR1402064.pdf Paper Reg. ID: IJEDR_140577, Published Paper ID: IJEDR1402064.

[21] Gary C Kessler. 2003. An overview of cryptography. (2003).

[22] Geir M Køien. 2015. A brief survey of nonces and nonce usage. *SECURWARE 2015* (2015), 97.

[23] Yung-Feng Lu, Chin-Fu Kuo, and Yi-Yen Fang. 2016. Efficient Storage Encryption for Android Mobile Devices. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems* (Odense, Denmark) *(RACS '16)*. Association for Computing Machinery, New York, NY, USA, 213–218. https://doi.org/10.1145/2987386.2987418

[24] Ueli M Maurer and Stefan Wolf. 2000. The diffie–hellman protocol. *Designs, Codes and Cryptography* 19, 2 (2000), 147–171.

[25] Mbed TLS Project. 2025. Mbed TLS Documentation. https://mbed-tls.readthedocs.io/en/latest/. Accessed: 2025-05-21.

[26] Khaled Salah Mohamed. 2020. *Cryptography Concepts: Confidentiality*. Springer International Publishing, Cham, 13–39. https://doi.org/10.1007/978-3-030-58996-7_2

[27] Samuel Neves. 2009. Faster ChaCha implementations for Intel processors. https://web.archive.org/web/20170328003903/https://cr.yp.to/chacha.html. Archived from the original on 2017-03-28, retrieved 2025-06-08. "Two of these constants are multiples of 8; this allows for a 1 instruction rotation in Core2 and later Intel CPUs using the `pshufb` instruction.".

[28] Takako Nonaka, Masato Shimano, Yuta Uesugi, and Tomohiro Hase. 2010. Embedded server and client system for home appliances on real-time operating system. In *2010 10th International Conference on Intelligent Systems Design and Applications*. 1345–1349. https://doi.org/10.1109/ISDA.2010.5687096

[29] Fabien A. P. Petitcolas. 2019. *Kerckhoffs' Principle*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–2. https://doi.org/10.1007/978-3-642-27739-9_487-2

[30] Gordon Procter. 2014. A Security Analysis of the Composition of ChaCha20 and Poly1305. Cryptology ePrint Archive, Paper 2014/613. https://eprint.iacr.org/2014/613

[31] Ronald L. RIVEST. 1990. CHAPTER 13 - Cryptography. In *Algorithms and Complexity*, JAN VAN LEEUWEN (Ed.). Elsevier, Amsterdam, 717–755. https://doi.org/10.1016/B978-0-444-88071-0.50018-7

[32] Richard Sharpe. 2002. Just what is SMB? *Oct* 8 (2002), 9.

[33] Satyajit Sinha. 2024. *Number of connected IoT devices growing 13% to 18.8 billion globally*. IoT Analytics. https://iot-analytics.com/number-connected-iot-devices/ Based on the "State of IoT Summer 2024" report.

[34] Rajeev Sobti and Ganesan Geetha. 2012. Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)* 9, 2 (2012), 461.

[35] William Stallings. 2010. *Cryptography and Network Security: Principles and Practice* (5th ed.). Pearson, Upper Saddle River, NJ.

[36] Rhys Weatherley. 2023. Ascon Suite - Lightweight Authenticated Ciphers. https://rweather.github.io/ascon-suite/asconcrypt.html Accessed: 2025-05-20.

[37] Wikipedia. 2025. Peer-to-peer file sharing. https://en.wikipedia.org/wiki/Peer-to-peer_file_sharing.

[38] Johannes Wilson and Mikael Asplund. 2025. Analysing TLS Implementations Using Full-Message Symbolic Execution. In *Secure IT Systems*, Leonardo Horn Iwaya, Liina Kamm, Leonardo Martucci, and Tobias Pulls (Eds.). Springer Nature Switzerland, Cham, 283–302.

[39] Wolf SSL Project. 2025. Wolf SSL Documentation. https://www.wolfssl.com/. Accessed: 2025-05-21.

[40] Anil Wurity and L. Sumalatha. 2024. ASCON: A New Era in Lightweight Cryptography. In *Advances in Cyber Security and Digital Forensics*. Iterative International Publishers (IIP). https://iipseries.org/assets/docupload/rsl202473E2C45E3E115D4.pdf e-ISBN.