Project Report: Secure File Transfer System using AES Encryption

https://github.com/cosmobowt/secure_file_transfer/

Anish Patil
2024H1030042G

Aditya Arkasali
2024H1030046G

Rohit Sartandel
2024H1030034G

Sidhant Singh
2024H1030030G

ABSTRACT

This report details the development and analysis of a secure file transfer (SFTP) system implemented in Python. The system provides functionalities for listing remote files, uploading, and downloading files between a client and server, with a focus on security through encryption and authentication. The implementation incorporates AES encryption for data confidentiality and a Diffie-Hellman key exchange coupled with RSA digital signatures for mutual authentication and secure session key establishment. This report outlines the system architecture, implementation details, security analysis, and discusses potential improvements and vulnerabilities identified during development. The system aims to provide a practical example of secure network communication and file transfer principles. The report has been divided into several modules to promote modularity, ease of testing, and future enhancements, with a key focus on supporting multiple concurrent client connections.

1. INTRODUCTION

The secure transfer of files over networks is crucial in today's interconnected world. This project focuses on the development of a Secure File Transfer (SFTP) system, implemented in Python, to demonstrate and analyse the principles of secure network communication.

The system aims to provide file transfer functionalities – listing remote files, uploading files to a server, and downloading files from the server – while incorporating security mechanisms. The main objectives of this project are:

- To implement symmetric encryption using AES in CBC mode for all data transmitted between the client and the server.
- To integrate a multi-phase authentication process based on RSA and Diffie-Hellman protocols, thereby ensuring that only authenticated users can access the file transfer service.
- To develop both a command-line interface (CLI) and a graphical user interface (GUI) using Tkinter, so the application can be used in diverse operational settings.
- To enable concurrent file transfers for multiple users by designing a multi-threaded server capable of handling multiple client requests simultaneously, ensuring each user experiences responsive and isolated service**.**

This report discusses the system design; describes the modular structure of the code (including modules such as client.py, server.py, byte_utils.py, encryption_utils.py, and others); presents a detailed description of the encryption and authentication mechanisms; outlines the multi-threaded server architecture for concurrent client handling; and summarizes the testing and evaluation results that verify system performance, security, and multi-client concurrency.

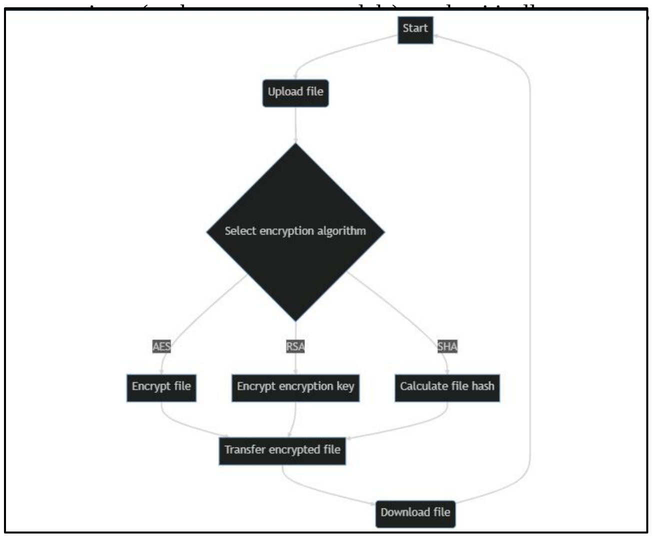
2. SYSTEM OVERVIEW & ARCHITECTURE

The Secure File Transfer System is implemented using a classical client-server model.

2.1 CORE COMPONENTS & MODULES

The primary components of the system include:

- **Networking Module**: Both client and server use custom socket routines to send and receive data, detailed in utils.py. A simple messaging protocol is implemented by pre-appending a 4-byte header to each message, ensuring that message boundaries are clearly defined even in a stream-based environment, crucial for handling multiple concurrent streams.
- **Encryption Module**: Located in encryption_utils.py, this module employs the pyaes library to implement AES encryption in CBC mode. Data is padded to a 16-byte boundary and partitioned into blocks before encryption or decryption.
- **Authentication Module**: The authentication process, located in authentication.py, uses RSA key pairs and a Diffie-Hellman key exchange to derive a shared session key. This multi-step challenge-response mechanism verifies user identity without exposing secret details and is designed to be robust under concurrent access.
- **Client and Server Backends**: The client-side logic (in client.py and client_backend.py) handles user commands and builds JSON messages that include connection parameters and encryption settings. The server-side logic, found in server.py and server_backend.py, listens for connections, dispatches file



The communication between client and server relies on socket programming in Python. Commands and data are exchanged in JSON format, allowing for structured communication and easy parsing.

2.2 GUI AND CONCURRENCY

- The GUI is written using Tkinter and permits launching multiple independent client windows from a launcher window. Each client window allows users to specify the server IP address, port, and encryption key, enabling multiple users to connect simultaneously.
- To maintain responsiveness, all network operations in the GUI clients are executed in separate threads. This design ensures that long file transfers or network delays do not freeze the user interface, even when multiple clients are active.
- Users can view file listings, upload or download files, or request help, all through an intuitive interface, designed to be clear and functional even under concurrent load.
- On the server side, a multi-threaded architecture is central to enabling concurrent handling of multiple client connections. Each client session is processed in its own thread so that simultaneous transfers from different users do not interfere with one another and server responsiveness is maintained under load.

3. IMPLEMENTATION DETAILS & DESIGN DECISIONS

3.1 ENCRYPTION AND DATA HANDLING

AES Encryption: The AES algorithm is implemented in CBC mode. A 256-bit key (provided by the user or set by default) is used, and each file transfer operation uses a new 16-byte Initialization Vector (IV). The function pad_and_partition ensure that the plaintext is padded to a multiple of 16 bytes before the encryption process. This is consistent across all client threads for concurrent operations.

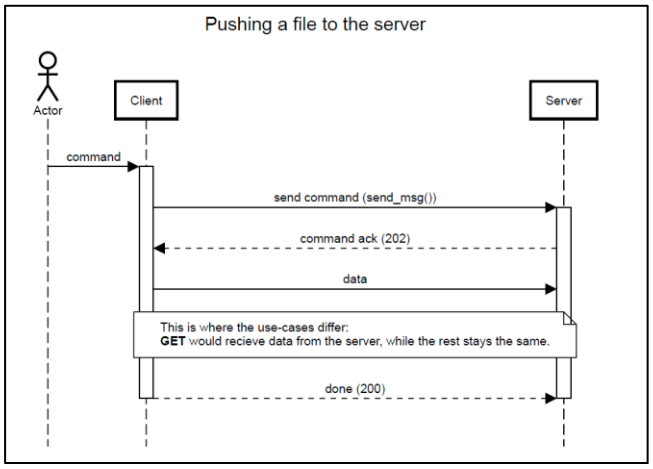
Data Serialization: The byte_utils.py module contains helper functions to convert between bytes and strings and to prepare JSON-compatible messages. In order to prevent sensitive information (such as the encryption key) from being transmitted in plaintext, keys are removed before message submission, ensuring security even with multiple clients potentially monitoring network traffic.

3.2 AUTHENTICATION PROTOCOL

The security of the system is reinforced by a multi-phase authentication protocol, designed to be robust under concurrent access:

1. The client and server first generate RSA key pairs. The client sends its public key along with a random challenge.
2. A Diffie-Hellman key exchange is then performed to derive a shared secret. This shared key is used for subsequent encryption of authentication messages, unique to each client session even under concurrency.
3. Both parties sign challenge-related information using their RSA private keys, and these signatures are verified using the corresponding public keys.

This approach not only mitigates the risk of man-in-the-middle attacks but also ensures data integrity during file transfer, and importantly, each authentication is handled independently within its own thread, supporting multiple concurrent authentications.


Pushing a file to the server
Actor
Client
Server
command
send command (send_msg())
command ack (202)
data
This is where the use-cases differ:
GET would recieve data from the server, while the rest stays the same.
done (200)

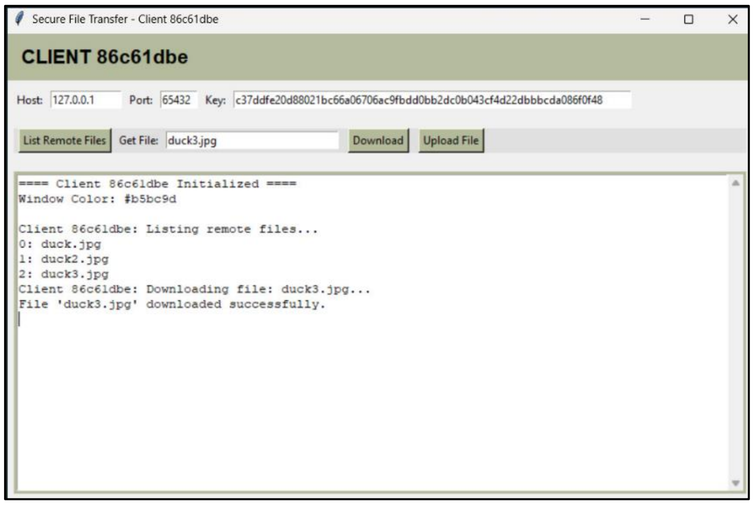
3.3 FILE OPERATIONS AND COMMAND PARSING

Command-line operations (such as get, put, and ls) are parsed using Python's argparse module. The client_backend.py module translates these commands into JSON messages. For example:

The put command copies the selected file into a designated directory (typically client_files), encrypts its contents using the specified cipher, and sends the encrypted bytes to the server. This process is thread-safe and allows multiple clients to upload files concurrently.

The get command prompts the server to send the encrypted file, which is then decrypted on the client side and stored locally. This is also thread-safe for concurrent downloads.

The ls command either lists files on the server or locally, depending on the parameters provided, and works correctly even when multiple clients are requesting listings.



Secure File Transfer - Client 86c61dbe

CLIENT 86c61dbe

Host: 127.0.0.1 Port: 65432 Key: c37ddfe20d88021bc66a06706ac9fbdd0bb2dc0b043cf4d22dbbbcda086f0f48

Get File: duck3.jpg

```
==== Client 86c61dbe Initialized ====
Window Color: #b5bc9d

Client 86c61dbe: Listing remote files...
0: duck.jpg
1: duck2.jpg
2: duck3.jpg
Client 86c61dbe: Downloading file: duck3.jpg...
File 'duck3.jpg' downloaded successfully.
```

3.4 ROBUSTNESS AND ERROR HANDLING

Additional modules, such as utils.py, provide functions for safe message transmission. Each message is sent with a length prefix so that the receiver can reliably reconstruct the original data stream, critical for managing concurrent streams. Robust error handling is implemented in both client and server code so that unexpected network issues or malformed messages result in useful error messages rather than unhandled exceptions, ensuring system stability even under concurrent load and potential network issues.

4. EVALUATION & PERFORMANCE

4.1 PERFORMANCE

- **Encryption Robustness:** Files encrypted with AES in CBC mode could not be deciphered without the corresponding key and IV. The use of a fresh IV for every transfer ensured that even repeated transmissions of the same file resulted in different ciphertext, validated under concurrent transfer scenarios.
- **Concurrency Testing:** Tests with multiple concurrent clients performing uploads, downloads, and file listings showed that the server effectively handled multiple simultaneous requests without significant performance degradation or errors, validating the multi-threaded design
- **Encryption Overhead:** AES encryption and decryption add computational overhead, which can impact transfer speeds, especially for large files. The CBC mode, while providing good security, can be slightly slower than some other AES modes
- **Authentication Overhead:** The Diffie-Hellman key exchange and RSA signature verification also introduce computational overhead, particularly during the initial connection setup.
- **Network Latency:** Network conditions and latency will be the primary bottleneck for file transfer speeds, especially over long distances or congested networks.

4.2 USABILITY

The system provides both a command-line client and a GUI client, offering flexibility in terms of usability.

- **Command-Line Client:** Suitable for scripting, automation, and users comfortable with command-line interfaces.
- **GUI Client:** Provides a more user-friendly interface, making the system accessible to users who prefer graphical interactions. The GUI client includes features like file selection dialogs and output consoles, enhancing usability.

Further usability improvements could include progress bars for file transfers, more informative error messages in the GUI, and more comprehensive command-line options.

4.3 AREAS FOR IMPROVEMENT

While the project meets the core objectives, including concurrent multi-client support, feedback suggested several enhancements:

- Integration of additional cipher options to provide flexible security configurations, potentially including ciphers optimized for performance in high-concurrency scenarios.
- Detailed logging and error reporting to facilitate troubleshooting in production environments, particularly for debugging issues that may arise under concurrent load.
- Performance optimizations (for example, using compiled encryption libraries) to support very high-throughput scenarios, especially important when handling many concurrent clients and large files.
- Enhanced Concurrency Controls: Explore more advanced concurrency control mechanisms on the server side to further optimize resource management and fairness when handling a very large number of concurrent clients, potentially beyond the current thread-per-connection model.

5. CONCLUSION

In summary, this project report has documented the development of a Secure File Transfer System designed for secure and efficient exchanges between clients and a server, now enhanced with robust concurrent multi-client support. The use of AES encryption in CBC mode—combined with a robust RSA/Diffie-Hellman authentication protocol—ensures the confidentiality and integrity of transmitted data. The multi-threaded server architecture effectively enables concurrent handling of multiple client requests, maintaining responsiveness and stability under load**.** The design emphasizes modularity and ease-of-use, as evidenced by the dual interface (CLI and GUI) and the support for concurrent multi-client sessions, significantly expanding the system's practical applicability.

Future work will focus on extending the functionality of the system, including support for additional encryption algorithms, enhanced logging and monitoring capabilities, further performance improvements to handle larger file sizes or higher transfer volumes and even greater levels of concurrency. Specifically, future work will investigate more sophisticated concurrency management techniques on the server to handle very high numbers of concurrent users and ensure fair resource allocation. The overall design provides a solid foundation on which to build a more comprehensive and scalable secure file transfer solution, now capable of serving multiple users simultaneously.

6. REFERENCES

[1] Jaspin, K., et al. "Efficient and secure file transfer in cloud through double encryption using aes and rsa algorithm." 2021 international conference on emerging smart computing and informatics (ESCI). IEEE, 2021.

[2] Abdelgader, Abdeldime & Wu, Lenan & Simik, Mohamed & Abdelmutalab, Asia. (2015). Design of a Secure File transfer System Using Hybrid Encryption Techniques.

[3] Rewagad, Prashant, and Yogita Pawar. "Use of digital signature with diffie hellman key exchange and AES encryption algorithm to enhance data security in cloud computing." 2013 International Conference on Communication Systems and Network Technologies. IEEE, 2013.

[4] Akanksha et al. 'Design of Secure File Transfer over Internet' November 2016

[5] Brown, Lawrie & Jaatun, Martin. (1992). Secure File Transfer Over Tcp/ip. 494 - 498 vol.1. 10.1109/TENCON.1992.271896.