Secure File Transfer Application

Juyoul Lee
Master of Science in Computer Information Systems
Florida Institute of Technology
juyoul2023@my.fit.edu
GitHub repository: https://github.com/ljy4499/secure-file-transfer-app

***Abstract*—The Secure File Transfer Application is designed to address the growing need for secure data exchange by utilizing modern cryptographic techniques. The application ensures confidentiality through AES encryption for file contents and secures key transmission using RSA encryption. Data integrity is maintained with SHA-256 hashing, and a user-friendly GUI simplifies the process of secure file transfer between systems. The application is versatile, supporting both local and public network environments.**

I. Introduction

As digital communication continues to evolve, the need for secure file transfer has grown significantly. Sensitive files must be protected against unauthorized access, tampering, and loss during transfer. It is equally important to ensure that only the intended recipient can access and decrypt the file.

This project focuses on developing a Secure File Transfer Application that addresses these challenges by combining symmetric and asymmetric encryption techniques. AES is used to encrypt file contents, ensuring confidentiality, while RSA facilitates the secure exchange of the AES encryption key. Additionally, SHA-256 hashing provides a mechanism to verify file integrity, preventing unauthorized alterations. Designed for ease of use, the application includes a graphical user interface and supports secure file exchanges across both local and public networks, making it suitable for a variety of environments.

II. Rationale for Algorithms Selection

The Secure File Transfer Application utilizes a combination of AES-CFB 256-bit encryption, RSA for key exchange, SHA-256 for file integrity verification, and WebSockets for communication. These technologies were selected based on their security, performance, and suitability for the project's requirements.

A. AES-CFB 256-bit with 32-byte Key

AES (Advanced Encryption Standard) with a 256-bit key is a widely adopted and highly secure symmetric encryption algorithm. The 256-bit key provides strong protection against brute-force attacks, ensuring robust confidentiality for file contents. AES in CFB (Cipher Feedback) mode was chosen over other modes like ECB (Electronic Codebook) or CBC (Cipher Block Chaining) due to its suitability for variable-length files. CFB mode is more flexible as it can encrypt partial blocks, which is crucial for file transfers where the size of the data is not fixed. Additionally, CFB mode prevents identical plaintext blocks from generating identical ciphertext blocks, thereby offering better protection against cryptanalysis.

B. RSA for Key Exchange

RSA (Rivest-Shamir-Adleman) is an asymmetric encryption algorithm used for securely transmitting the AES key. In this project, RSA allows the sender to encrypt the AES key using the recipient's public key, ensuring that only the recipient, with their private key, can decrypt the AES key. This eliminates the need for a pre-shared secret, enhancing security by mitigating risks associated with key distribution and management. RSA provides a reliable mechanism for secure key exchange, facilitating safe transmission of the AES key over insecure networks.

C. SHA-256 for File Integrity Verification

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function chosen for verifying the integrity of files during transfer. SHA-256 produces a unique 256-bit hash value for each file, ensuring that any modification to the file during transmission can be detected by comparing the hash values before and after transfer. This provides an additional layer of security, ensuring that files remain untampered with and are transmitted without corruption.

D. WebSockets for Communication

WebSockets were chosen over FTP for their efficiency and simplicity in real-time, bidirectional communication. Unlike FTP, which requires multiple connections and is more complex to configure, WebSockets provide a persistent, low-latency connection, making file transfer faster and more reliable. They also integrate seamlessly with cryptographic algorithms such as AES, RSA, and SHA-256, allowing for straightforward encryption and secure data transfer.

WebSockets also support secure communication via SSL/TLS, ensuring encrypted data transfer across both local and public networks. In contrast, FTP requires additional configurations for secure transfer (e.g., FTPS), making WebSockets a more flexible and customizable solution for secure file exchange.

III. Flowchart

The workflow of the Secure File Transfer Application is illustrated in Figure 1. The process begins with the receiver opening a port to run the server. On the sender's side, the user selects a file for transfer. The file is then hashed using the SHA-256 algorithm to ensure data integrity. Next, the file is encrypted using AES in CFB mode with a 256-bit key. The 32-byte AES encryption key is then secured by encrypting it with RSA using the receiver's public key.

After the encryption process, the sender transmits the encrypted file, the file's hash, and the RSA-encrypted AES key to the receiving system via TCP sockets. On the receiver's end, the RSA-encrypted AES key is decrypted using the receiver's private key to obtain the AES key. This key is then used to decrypt the file. Finally, the integrity of the received file is verified by comparing its SHA-256 hash with the hash computed at the sender's side.

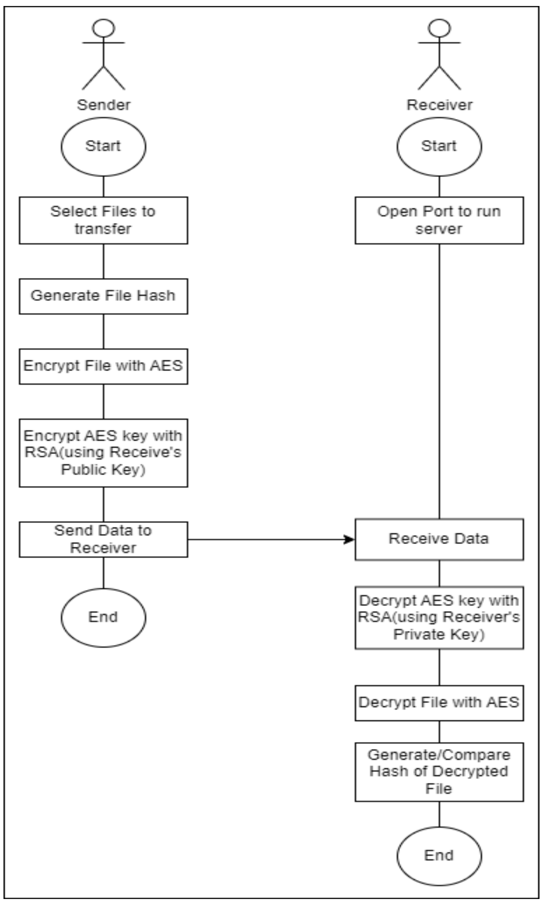

Fig. 1. Flowchart of the Secure File Transfer Application

IV. Application GUI

The application incorporates a GUI to enhance user experience. The interface is divided into two primary sections: one for sending files and another for receiving files. The sending section allows users to select a file, encrypt it, and transfer it to a specified recipient. The receiving section facilitates the decryption and integrity verification of received files. Screenshots of these sections are provided in Figures 2

The debugging terminal displays key information during file transfer. It logs the server's listening status, the file's SHA-256

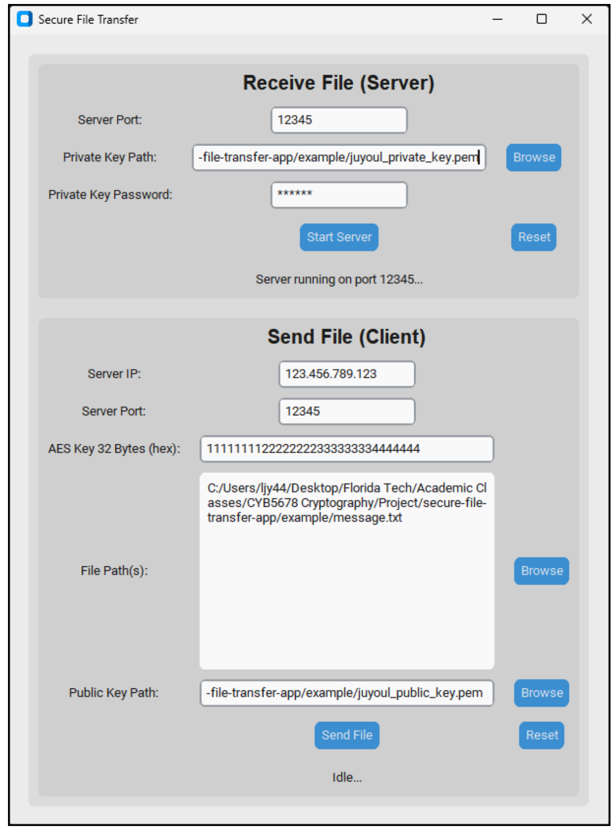

Fig. 2. GUI for Sending and Receiving Files

hash calculation, and AES encryption details. Additionally, it shows the lengths and previews of the encrypted file and AES key, along with connection details. After decryption, the terminal verifies the file's integrity by comparing hashes and confirms successful receipt and decryption. This ensures transparency and traceability throughout the process.

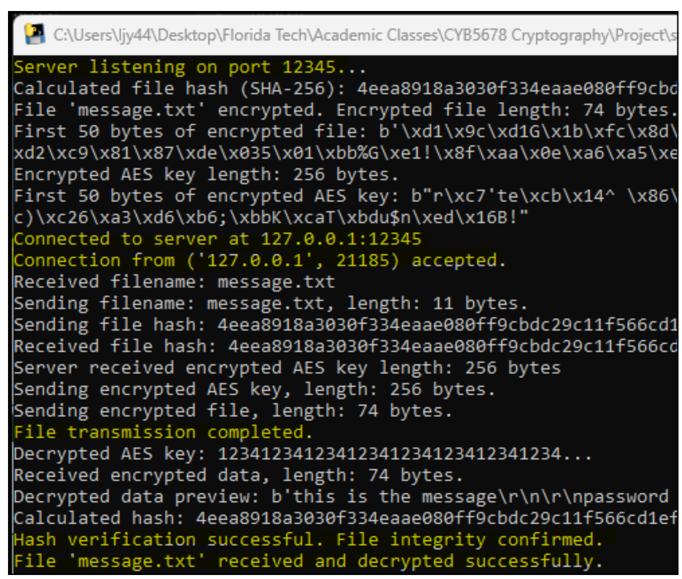

Fig. 3. Terminal for Debugging

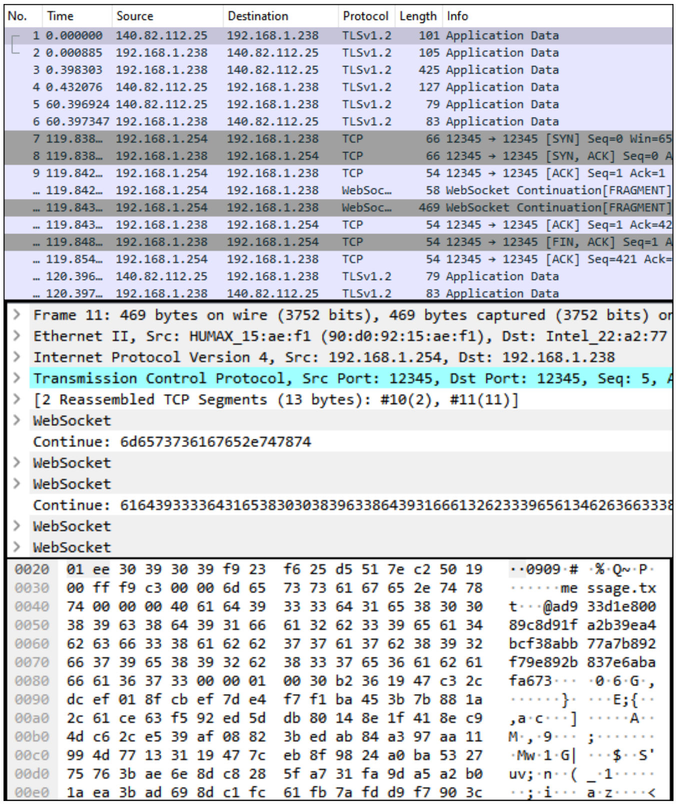

Fig. 4. Packet Capture on Wireshark

V. Key Learnings

This project provided valuable insights into the practical application of cryptographic techniques. Implementing AES and RSA encryption deepened our understanding of symmetric and asymmetric encryption methods, especially in the context of secure file transfer. Working with SHA-256 hashing also highlighted the importance of data integrity in maintaining secure communication.

The project exposed the complexities of public key distribution. In this setting, the sender already has the receiver's public key, but in a real-world application, secure distribution using Certificate Authorities (CAs) would be essential. Additionally, performance optimization was a key learning point, emphasizing the need to balance the encryption algorithms—AES for file encryption and RSA for key exchange—to ensure both security and optimal performance during file transfers.

VI. Challenges

During the development of the Secure File Transfer Application, several challenges were encountered. Ensuring data integrity and ensuring the receiver gets the exact data sent by the sender was initially problematic, with issues of broken data due to misconfigurations. These were resolved by refining the transmission process. Another challenge was managing the transfer of multiple files in a single session instead of transferring them individually. This was also solved by implementing multi-file handling capabilities.

A critical challenge that remains unsolved is the implementation of a feature for the receiver to validate the file after receiving its metadata. This would allow the receiver to accept or reject a file based on security concerns regarding potentially malicious files.

VII. Future Recommendations

To further enhance the Secure File Transfer Application, several improvements can be made:

- **Advanced Algorithms**: Incorporate Diffie-Hellman key exchange or TLS for more secure key exchange mechanisms, and consider integrating Post-Quantum Cryptography to future-proof the application against emerging threats.
- **Post-Transfer Verification**: Implement a feature allowing the receiver to accept or deny file transfers after validating the file metadata to prevent the acceptance of malicious files.
- **Multi-Platform Compatibility**: Extend support for Linux, macOS, and mobile devices through cross-platform frameworks, ensuring broader accessibility.
- **Server-Client Authentication**: Integrate user authentication with role-based access control (RBAC) and encrypted metadata exchange to ensure secure interactions between users and systems.
- **PKI for Key Distribution**: Implement Public Key Infrastructure (PKI) to securely manage and distribute public keys using digital certificates.

VIII. Conclusion

The Secure File Transfer Application successfully integrates AES, RSA, and SHA-256 to provide a robust solution for secure file sharing. The project not only addresses the confidentiality and integrity requirements of secure file transfer but also considers the importance of performance optimization. While some challenges remain, such as implementing file validation and secure public key distribution, the application provides a strong foundation for further enhancements. The recommendations for integrating advanced cryptographic techniques, improving platform compatibility, and enhancing security features will help future-proof the application for real-world use.

References

[1] "Symmetric encryption," Cryptography, Accessed: Dec. 6, 2024. [Online]. Available: https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/.

[2] "RSA — Cryptography documentation," The Cryptography Library, Accessed: Dec. 06, 2024. [Online]. Available: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/.

[3] F. Khan, "Modern GUI using Tkinter," Medium, Sep. 4, 2022. [Online]. Available: https://medium.com/@fareedkhandev/modern-gui-using-tkinter-12da0b983e22