CISC 468 P2P Secure File Sharing Application

Alex Panfilov (20301313, alex.panfilov@queensu.ca)
Alex Dobrovolskyy (20298057, 21asd4@queensu.ca)

ABSTRACT

The purpose of this report is to provide a comprehensive overview of the design and implementation of a secure peer-to-peer (P2P) file sharing application. This application aims to facilitate safe and efficient file sharing between users within a local network, fulfilling various critical objectives focused on security and usability, despite the user clients being implemented with different cryptographic APIs in two different languages (Python and Java).

1 INTRODUCTION

The project presented in this report focuses on the development of a secure P2P file-sharing application designed specifically for safe and reliable communications within local networks. One of the primary objectives is to implement a user-friendly interface while ensuring robust security features that protect shared files from potential threats. The application is built using Java and Python, implementing different cryptographic APIs in two different languages.

Key features of the application include mDNS-based peer discovery, which simplifies the process of locating other users on the network, mutual authentication that ensures that users can verify each other's identities, key rotation, and file transfer functionality. To protect the confidentiality and integrity of the data being exchanged, the application employs encryption techniques such as AES-256-CBC and ephemeral Diffie-Hellman session keys, which ensure the system supports perfect forward secrecy. Due to this, even if a user's private key is compromised, past communications remain secure.

By focusing on user experience alongside security measures, this project aims to make the system easy to use for both coding novices and experts. It is intuitive to use due to the labeling and grouping of all the functionalities available.

2 APPLICATION DESIGN OBJECTIVES

The design objectives for the secure P2P file-sharing application focus on creating a robust, secure system. Key objectives include:

2.1 Peer Discovery on a Local Network.

The application supports peer discovery on a local network using mDNS, allowing devices to dynamically find and communicate with each other. By registering services and detecting peer IP addresses and ports, mDNS enables seamless connectivity without requiring a central server. This approach was chosen for its lightweight nature, ease of implementation, and ability to facilitate direct device-to-device communication without additional configuration. Using JmDNS in Java and Zeroconf in Python, each peer can register its service with a digital fingerprint and a dynamic port, ensuring that no central server is required.

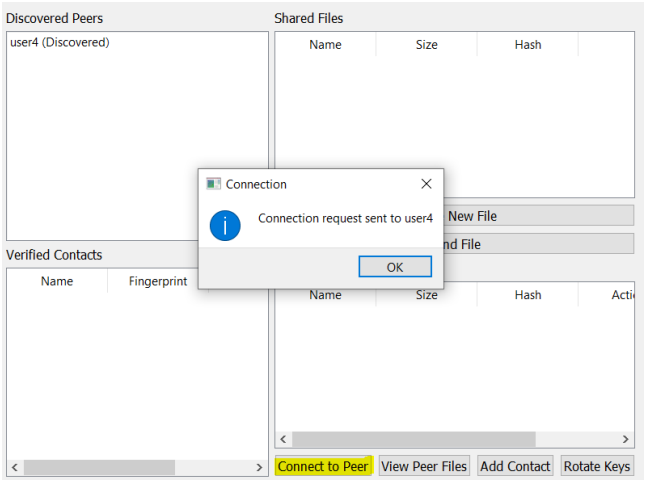

Figure 1: Sent connection request to discovered peer.

2.2 Mutual Authentication of Contacts

To ensure secure interactions, the application implements mutual authentication through cryptographic fingerprint verification, ensuring both parties validate each other's identities before establishing a connection. During the handshake process, peers exchange fingerprints - unique identifiers derived from the Diffie-Hellman key exchange - embedded in connection_request and connection_response messages. The initiating peer sends its fingerprint in the request, which the recipient verifies against stored or pre-shared data. If valid, the recipient responds with its own fingerprint, which the initiator similarly verifies. After both fingerprints are confirmed, the connection is marked as trusted and added to connected_peers.

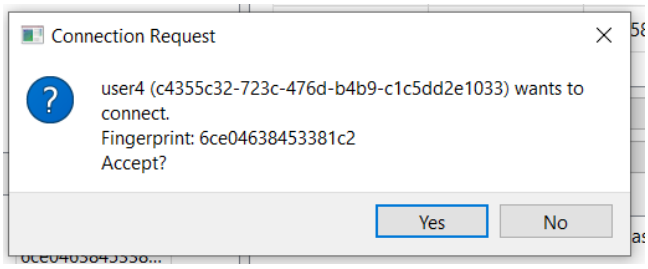

Figure 2: Connection request from peer that user can accept or reject.

2.3 File Transfer Requests

The application enforces user consent for file requests through GUI-based prompts. When a peer requests a file, the receiving user (if

request was from a verified peer) sends the file automatically. on the other hand, if the request is from a non verified peer, the receiver is presented with a message showing that a unverified peer is trying to access there files. This mechanism enhances security and user control by ensuring that file-sharing activities remain intentional and that sensitive data is not accessed without explicit permission.

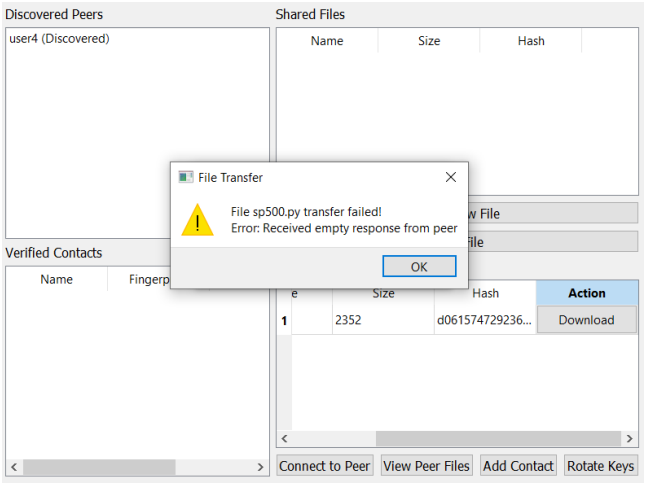

Figure 3: An error message due to attempt to download file from unverified peer.

2.4 File List Requests

For ease of collaboration, the application allows peers to request and receive file listings without requiring consent if they are both registered as verified contacts. Using socket-based communication, users can view available files on other devices while still maintaining security by enforcing consent when an unverified peer attempts to access a file listing. This functionality simplifies resource discovery while ensuring that access to files remains controlled.

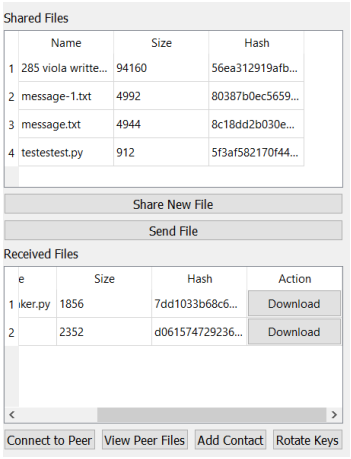

Figure 4: Shared and received file lists from a user.

2.5 Fallback Mechanism for Offline Peers

One of the goals was to ensure that if peer A goes offline, another peer B can still get files by finding another peer C that has previously downloaded those files. The integrity of the files must be ensured through verification mechanisms to confirm that the downloaded file is identical to the one originally offered by peer A. We have mechanisms to detect whether the expected hash matches the actual calculated hash of a file, before download of the file commences. We were not able to implement the matching of peer A's files to a user C which has already downloaded said files, for the purpose of allowing user B to download these files. This was due to the difficulty of testing such a scenario with only two local machines. This remains a requirement that must be fulfilled in future work.

2.6 Key Migration for Compromised Keys

To handle key compromises, the application supports key migration, allowing users to generate a new key by pressing the button "Rotate Keys", which generates a new secure key for the user and notifies their contacts of the change through a GUI popup. Once the contacts are notified, their clients automatically replace the compromised key with the new key of their peer.

```
Connection status changed: peer_id=66047ed6-dd9b-4b23-8914-b106db1375c4, connected=True
Received key change notification from user20 (66047ed6-dd9b-4b23-8914-b106db1375c4): ec12080e9824c662
Updated fingerprint for 66047ed6-dd9b-4b23-8914-b106db1375c4
Connection status changed: peer_id=66047ed6-dd9b-4b23-8914-b106db1375c4, connected=True
```

Figure 5: Logging output stating that peer's key change and updating fingerprint for peer.

2.7 File Confidentiality and Integrity

Confidentiality is ensured through AES-256-CBC encryption and decryption with session-specific keys derived from ephemeral ECDH (SECP384R1) key exchanges, where fresh keys and random IVs are generated for each file transfer to prevent retroactive decryption. Integrity is enforced using SHA-256 file hashes to verify data post-decryption and mutual authentication via cryptographic fingerprints exchanged during handshakes, ensuring peers validate each other's identities. File download and decryption can be blocked if the file integrity is violated.

2.8 Perfect Forward Secrecy

Perfect forward secrecy is implemented through ephemeral Elliptic Curve Diffie-Hellman (ECDH) key exchange, ensuring each file transfer session uses cryptographically independent keys. When a file transfer is initiated, both peers generate fresh ephemeral ECDH key pairs (using the SECP384R1 curve) and exchange public keys in DER format (base64-encoded). A shared secret is derived via ECDH, hashed via SHA-256 to create a 256-bit AES session key, and combined with a randomly generated 16-byte IV for AES-CBC encryption. This process guarantees that even if a node's long-term authentication keys are compromised, prior file transfers remain secure, as ephemeral private keys are discarded after use, hence guaranteeing perfect forward secrecy.

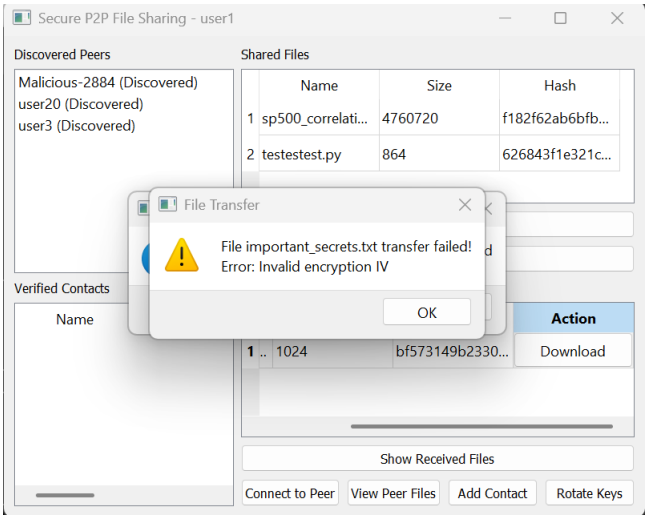

Figure 6: An error message of invalid encryption IV because the file was tampered with.

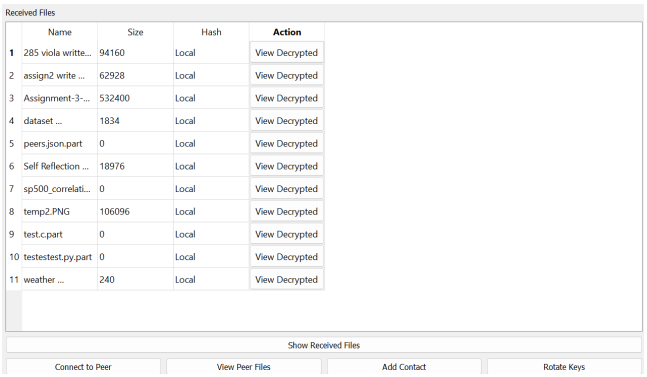

Figure 7: List of downloaded files that can be decrypted and viewed.

2.9 Secure Local File Storage

After successfully decrypting and unpadding the data downloaded from a peer, the plaintext is written to a temporary file on disk. Immediately after, for secure local storage, the file is re-encrypted using a locally derived key. This key is generated by hashing the device's fingerprint using SHA-256, which produces a strong cryptographic key. By encrypting stored files, unauthorized access is prevented, protecting sensitive data in scenarios where physical security may be compromised.

2.10 Error Handling and User Notifications

The application provides user notifications for errors and security-related issues through GUI alerts and logging functionality. Users receive feedback when operations fail or security verifications do not pass. This proactive notification system allows users to take corrective actions when necessary.

3 ADDITIONAL FUNCTIONALITY

In our system, files were initially downloaded and decrypted during transfer; however, to ensure secure local storage, each file was subsequently re-encrypted using a locally derived key, which was derived from the user's fingerprint. While this approach effectively safeguarded sensitive files in case the device was stolen, it introduced usability limitations, as users would require an external decryption tool to manually access and view their stored files.

To address this issue and improve usability without compromising security, we extended the Python client's functionality by integrating a user-accessible decryption interface. This enhancement introduced a graphical interface with a "Show Received Filed" button that lists all encrypted files stored locally. Each file entry was accompanied by a "View Decrypted" button, which, when clicked, would securely decrypt the corresponding file in memory and open it using the appropriate viewer. This solution preserved confidentiality while offering seamless access to the user, eliminating the need for external tools or manual decryption.

4 LIBRARIES, ALGORITHMS, AND SECURITY PARAMETERS

Outlined below are the libraries, algorithms, and security parameters which were utilized in our application.

- mDNS libraries JmDNS (Java) and Zeroconf (Python) are used for automatic peer discovery within a local network.
- Python used the cryptography and socket libraries, whereas Java used crypto and security libraries for file transfer and encryption algorithms.
- Elliptic Curve Diffie-Hellman (ECDH) protocol is used for secure key exchange between peers. ECDH allows both parties to establish a shared secret over an insecure channel, providing strong encryption and better performance than traditional Diffie-Hellman. We used P-384 curve (ec.SECP384R1()), as it is supported by most clients and is more secure that P-256 curve.
- AES-256-CBC is used for encrypting and decrypting files during transfer. This symmetric encryption algorithm with a 256-bit key ensures strong data protection, and the CBC mode adds complexity by chaining encrypted blocks, increasing resistance to attacks.

5 CHALLENGES FACED

This section explains some of the main challenges we encountered throughout the development.

5.1 Testing Challenges

One of the main challenges we encountered was the need to work in close proximity to effectively test peer-to-peer functionality, especially the initial client discovery implementation. Since every feature required devices to be on the same network and actively communicating, it was difficult to make progress without being in the same physical space. This limitation not only slowed down

development but also made debugging and validating our solutions more complex.

5.2 Connection Issues

We initially encountered difficulties connecting clients across different machines, as they were unable to discover one another despite successful communication between multiple users on the same device. Upon investigation, we found that the issue stemmed from incorrect configuration settings, which prevented proper peer discovery over the local network. Once these configuration parameters were corrected - ensuring that mDNS traffic could propagate - clients on different machines were able to successfully discover and connect with each other.

5.3 Difficulties with GUI

During the GUI implementation using PyQt5, we encountered several challenges related to the complexity of integrating asynchronous networking and cryptographic operations within a responsive user interface. PyQt5 was chosen due to its popularity and extensive documentation, as well as the advanced interface components it could provide in comparison to another popular graphics library, Tkinter.

However, PyQt5 is by default single-threaded, so long-running operations such as peer discovery or file transfers made the GUI freeze. To overcome this issue, we implemented QThread, which allowed us to separate the front-end and back-end functionality and hence provide the user with a responsive GUI. Implementing the GUI in Java presented challenges primarily due to Java's more rigid and verbose syntax compared to Python. This stricter structure made the process of creating and modifying GUI components significantly more time-consuming.

5.4 Challenges with Interoperability

In the initial implementation of the key exchange mechanism, cryptographic keys were exchanged using files encoded in the PEM (Privacy-Enhanced Mail) format. This format was also used for transmitting the shared secret during file sharing. However, a significant limitation arose when integrating with the Java client: Java lacks native support for parsing PEM-formatted keys without relying on external libraries or additional parsing logic. While we initially attempted writing custom logic on the Java side to support PEM parsing, we ultimately opted to switch to the DER (Distinguished Encoding Rules) format. DER, a binary representation of the same data used in PEM, does not include headers or human-readable statements, making it more compact and easier to parse. This change eliminated the need for extra parsing logic and improved cross-platform interoperability, allowing seamless and reliable communication between Python and Java clients.

6 CONCLUSION

The secure P2P file sharing application, developed with interoperable Java and Python clients, successfully achieves most of its design objectives. By implementing critical security measures including mutual authentication, permission-based file transfers, and perfect forward secrecy, the system establishes a reliable foundation for protected communications. The application has successfully demonstrated cross-platform interoperability through effective client-to-client file sharing operations.

For future development, we plan to implement three key improvements: 1) comprehensive offline functionality that maintains file integrity when accessing documents from disconnected peers, 2) user interface refinements to enhance overall usability, and 3) decrypted file viewing capability within the Java client. These enhancements will further strengthen the application's functionality while maintaining its security standards.