# SECURE FILE TRANSFER SYSTEM USING HYBRID AES-RSA CRYPTOGRAPHY AND ROLE-BASED ACCESS CONTROL

MRS. K. PAVANI[1], K. KAVYA[2]

[1]HOD & Assistant Professor, Department of Master of Computer Applications, SRK Institute of Technology, Vijayawada, Andhra Pradesh

[2]MCA Student, Department of Master of Computer Applications, SRK Institute of Technology, Vijayawada, Andhra Pradesh

## ABSTRACT

The exchange of digital files across computer networks has become a routine activity in enterprise, healthcare, academic and governmental settings. Conventional mechanisms such as the File Transfer Protocol (FTP) and unmodified TCP socket programs transmit credentials and content in cleartext, leaving sensitive information exposed to passive interception on shared infrastructure. This paper presents the design and implementation of a Secure File Transfer System (SFTS), built entirely in Python, that addresses these vulnerabilities through a multi-layered cryptographic architecture. The system combines AES-256-CBC symmetric encryption of bulk file data with RSA-2048-OAEP asymmetric key wrapping so that the per-session AES key never appears on the wire in plaintext form. Data integrity is verified by SHA-256 checksums embedded in every encrypted payload. Two delivery modes are supported: a reliable unicast mode over TCP and a one-to-many broadcast mode over UDP, the latter augmented with a TCP feedback channel for targeted retransmission of missing chunks. User authentication is performed by a dedicated module that stores bcrypt-hashed passwords in an SQLite database and enforces three roles — administrator, user and blocked — at login time. The system exposes its functionality through both a PyQt5 desktop interface and a Flask web dashboard. Experimental measurements on a local area network demonstrate sustained throughput of approximately 19 MB/s for unicast transfers up to 100 MB, while security testing confirms that the cryptographic guarantees hold against passive eavesdropping, ciphertext tampering and credential-based bypass attempts.

**Keywords:** AES-256-CBC, RSA-2048-OAEP, Hybrid Cryptography, Secure File Transfer, bcrypt, Role-Based Access Control, SHA-256 Integrity, Audit Logging, Python, PyQt5, Flask.

## 1. INTRODUCTION

### 1.1 Overview

Digital file exchange is among the most frequent operations performed on contemporary computer networks. Hospitals share patient records between workstations, universities distribute examination materials, government departments transfer policy documents and software development teams continuously move source artifacts between machines. Each of these workflows depends on a transport that is assumed to preserve the confidentiality of the data and to deliver it intact. The earliest protocol designed for this purpose, the File Transfer Protocol introduced in RFC 959, predates the modern threat model by several decades. FTP transmits user names, passwords and file content as plain bytes, making any party with packet-capture access an effective adversary. Even today, FTP and bespoke TCP socket programs remain in use within laboratories, small enterprises and academic projects, often because their setup overhead is low and their operation is intuitive.

The cost of this convenience is substantial. Within shared wireless segments, college LANs and small office networks, a single workstation running a sniffer can recover transmitted credentials and reconstruct entire files. Mature alternatives such as FTPS, SCP and SFTP do encrypt the channel, but they delegate every cryptographic operation to large, opaque TLS or SSH libraries written in compiled languages. While that delegation is the correct engineering decision for production-grade deployment, it conceals the security choices from the developer and student. As a result, the cryptographic reasoning behind a secure transfer protocol — why an initialization vector must be random, why a session key must be wrapped, how integrity is detected — is rarely visible at the application layer.

The work described in this paper takes a different approach. Rather than rely on a delegated transport library, it implements every cryptographic operation explicitly in Python at the application layer. The result is a self-contained Secure File Transfer System (SFTS) that fuses AES-256 bulk encryption with RSA-2048 session-key wrapping, SHA-256 payload digests, bcrypt-protected user credentials and a structured audit trail within a single coherent codebase. The system offers both a PyQt5 desktop interface and a Flask-based web dashboard, supports both reliable unicast and reliable broadcast transfer modes, and persists all relevant state in lightweight SQLite databases.

### 1.2 Motivation

The motivation for this work has three strands. First, no widely used secure-transfer tool combines a transparent Python implementation with both desktop and web management interfaces. OpenSSH SFTP is a command-line tool with no graphical front-end; FileZilla and WinSCP are graphical clients but neither exposes its cryptographic core to inspection. For an academic or training setting in which a student is expected to understand the protocol rather than treat it as a black box, none of these is suitable. Second, many institutional environments cannot or will not deploy a full-scale SSH or TLS server simply to permit secure file exchange within a single LAN. A self-contained Python application that runs from a project folder, persists its state in local SQLite files and listens on a configurable TCP port is dramatically easier to deploy and administer in such settings. Third, repeated data-breach incidents in recent years have demonstrated that file transfer mechanisms which provide encryption only as an opt-in configuration are insufficient. A modern design must make cryptographic protection the default and unconditional behaviour of the transfer operation.

### 1.3 Problem Statement

The central problem addressed by this work is to design and implement a self-contained file transfer application that simul-

taneously satisfies a number of properties that conventional tools deliver only partially or not at all. Specifically, the system must encrypt every file in transit using a contemporary symmetric cipher; exchange the corresponding session key without exposing it on the wire; detect every form of payload tampering or accidental corruption before the recipient acts on the received data; authenticate every user through a password-based scheme that resists offline brute-force attack; enforce a role-based authorisation policy on all administrative operations; produce a persistent, structured record of every transfer event; and support both point-to-point and one-to-many delivery within a local area network. All of these requirements must be met without depending on any external service or pre-existing public-key infrastructure.

### 1.4 Applications

The system is suitable for a range of practical scenarios:

- Secure exchange of patient health records between workstations inside a hospital LAN where regulatory requirements demand confidentiality of transmitted data.
- Distribution of confidential examination papers, evaluation rubrics and award lists between staff machines in an academic institution.
- Internal document transfer for small and medium enterprises that lack a dedicated VPN or document management server.
- Educational laboratories in which students study cryptographic protocols in operation and inspect each step of the key-exchange and encryption pipeline.
- Inter-departmental document transfer in government offices operating on physically isolated networks.
- Secure distribution of source code, build artefacts and configuration files within small research and development teams.

## 2. LITERATURE REVIEW

A substantial body of work has shaped the modern landscape of secure file transfer. The original File Transfer Protocol [1] was standardised by Postel and Reynolds in 1985 and was, by design, unconcerned with confidentiality. To remedy this, the Internet Engineering Task Force introduced FTPS [2], which wraps both the control and data channels of FTP in Transport Layer Security. While FTPS preserves the operational semantics of classical FTP, the requirement to negotiate TLS on two separate channels has historically complicated its interaction with stateful firewalls and Network Address Translation.

The Secure Shell family of protocols offers an alternative route. SCP, the simpler of the two SSH-based file-transfer tools, provides confidentiality and integrity for one-shot copies but lacks support for resumable transfers and fine-grained access control. SFTP [3], formally defined as a subsystem of SSH, is considerably more capable: it supports directory listing, file-attribute lookup, fractional read and write operations and byte-range locks. The reference OpenSSH implementation runs to tens of thousands of lines of C and is engineered for production deployment rather than pedagogical clarity.

The cryptographic primitives that underpin all of the above protocols are themselves well studied. Stallings [4] provides a textbook treatment of hybrid cryptography, establishing that the combination of a fast symmetric cipher for bulk data with a slower asymmetric cipher for key wrapping forms the foundation of essentially every modern secure transport. Schneier [5] enumerates the practical pitfalls of Cipher Block Chaining mode, emphasising that an attacker who can predict or replay an initialisation vector can compromise confidentiality even when the underlying block cipher is sound.

The choice of asymmetric algorithm has been an active area of evolution. RSA, originally proposed by Rivest, Shamir and Adleman in 1978, remains widely deployed in 2048-bit form; the PKCS#1 v2.1 specification [6] introduced the Optimal Asymmetric Encryption Padding scheme to defeat malleability attacks. In the authentication domain, the bcrypt scheme of Provos and Mazières [7] introduced a memory-hard password hashing function with a configurable cost parameter, specifically designed to defeat brute-force speedups available on commodity graphics hardware.

Audit logging has received less attention in the academic literature but is a regulatory requirement under several frameworks, including HIPAA §164.312(b) and PCI-DSS Requirement 10. Lioy and Ramunno [8] argued for in-application logging at the protocol layer, observing that operating-system logs alone are too coarse to capture the semantic events of an application-level transfer.

Several recent works have examined Python-based secure transfer implementations. Prakash and Saravanan [9] reported that a hybrid AES-256 plus RSA-2048 design implemented in Python achieves throughput comparable to unencrypted baselines for files up to one hundred megabytes. Yadav et al. [10] evaluated five Python cryptographic libraries, finding PyCryptodome to be the most suitable for application-layer designs. Gupta and Sharma [18] reported that SQLite in Write-Ahead Logging mode achieves approximately thirty per cent higher write throughput under mixed concurrent workloads, motivating the WAL configuration adopted here. Verma and Tiwari [19] demonstrated that chunked UDP delivery with selective retransmission can achieve effective reliability without the connection-establishment overhead of TCP.

A useful summary comparison of the principal protocols and their properties is given in Table 2.1. The comparison highlights that every modern alternative to plain FTP delegates its cryptographic work to a compiled library; none exposes its key-management decisions to the application developer; and none combines reliable broadcast with strong authentication out of the box.

| Tool | Channel | Symmetric | Auth | GUI | Audit | Lang. |
|---|---|---|---|---|---|---|
| Plain FTP | TCP | none | plain-text | optiona | external | varies |
| FTPS | TLS | AES/ChaCha | cert/ pwd | optiona | external | C |
| SCP | SSH | AES/ChaCha | key/ pwd | none | external | C |
| OpenSSH SFTP | SSH | AES/ChaCha | key/ pwd | none | OS log | C |
| Pro-posed SFTS | TCP/UDP | AES-256-bc CBC | pt+RB | Cdesk op+wel | SQLite | Python |

*Table 2.1: Comparison of secure file-transfer mechanisms.*

While each of the primitives used in the present work is independently mature, no readily available open-source implementation in Python fuses these primitives into an end-to-end application that exposes both desktop and browser control surfaces, supports both point-to-point and group delivery, and persists a complete operational log.

## 3. PROPOSED SYSTEM

### 3.1 System Overview

The proposed Secure File Transfer System is organised as a set of cooperating modules running on the participating machines. Each machine can operate concurrently as a sender, a receiver or both, and can host either the desktop application, the web dashboard, or both. The system depends on no external server, no central authority and no pre-existing public-key infrastructure beyond a single optional broadcast keypair that is generated on first run.

At the highest level, the architecture is partitioned into five concerns: a user-interface layer comprising the PyQt5 desktop application and the Flask web dashboard; an authentication-and-authorisation layer implemented by the auth module; a transfer layer comprising independent unicast and broadcast sender and receiver modules; a cryptography layer encapsulated in the encryptor module; and a persistence layer realised by four lightweight SQLite databases for user records, audit entries, configuration values and resumable transfer state respectively.

### 3.2 Cryptographic Design

The cryptographic core of the system is a hybrid scheme that combines symmetric and asymmetric primitives. For each transfer, the sender generates a fresh 32-byte AES-256 session key using a cryptographically secure pseudorandom number generator. The plaintext file is encrypted under this key in Cipher Block Chaining mode with PKCS#7 padding and a freshly generated 16-byte initialisation vector. A SHA-256 digest of the original plaintext is computed before encryption and is stored alongside the IV and ciphertext. The resulting encrypted payload therefore has the layout IV ‖ SHA-256(plaintext) ‖ AES-CBC ciphertext.

For unicast transfers, the receiver generates an ephemeral RSA-2048 keypair at the start of each session and transmits the public component to the sender. The sender wraps the AES session key using PKCS#1 v2.1 Optimal Asymmetric Encryption Padding under the receiver's public key. Once the file has been delivered, the receiver discards both the AES key and the RSA private key, yielding forward secrecy with respect to that session. A passive eavesdropper who captures the entire TCP exchange recovers only a 256-byte OAEP ciphertext where the AES key would otherwise have appeared in cleartext.

For broadcast transfers, the constraints of one-to-many UDP delivery make individual key exchange impractical. The system therefore relies on a pre-shared broadcast keypair generated during system initialisation and stored in PEM format. Every authorised receiver holds the broadcast private key. The sender wraps the per-transfer AES key with the broadcast public key and embeds the resulting ciphertext in the metadata header of the broadcast.

### 3.3 Authentication and Authorisation

User accounts are managed by a dedicated authentication module that backs onto an SQLite database with a single users table. Each row holds a unique username, a bcrypt hash of the user's password, the user's role and the time of account creation. The bcrypt hash incorporates a randomly generated salt per registration and a configurable work factor, so two users sharing the same plaintext password store entirely different hash strings.

The system defines three roles. Administrators may register, block and unblock other accounts; modify the file-type whitelist and the maximum file-size policy; and view the audit log for any user. Standard users may perform transfers within the configured policy and may view their own audit history. The blocked role is a soft-deletion mechanism: the account continues to exist for audit-trail integrity but is denied login. A safeguard in the role-update routine prevents an administrator account from being demoted to the blocked role.

### 3.4 Audit Logging Framework

Every transfer attempt generates an entry in a separate audit-log SQLite database. The entry records ten fields: a sequential identifier, a high-resolution timestamp, the username that initiated the operation, the action, the file name, the file size in bytes, the transfer mode, the peer IP address, a status code and a free-form details field. The schema satisfies the Who-What-When-Where-How-Result framework recommended by audit-compliance frameworks. Access to the audit database is mediated by a role-aware query function: administrator accounts receive all entries; standard users receive only their own activity.

### 3.5 Reliability for Broadcast Transfers

Native UDP broadcast offers no delivery guarantees: packets may be dropped, duplicated or reordered. To compensate, the broadcast sender partitions the encrypted payload into indexed chunks of up to 8,187 bytes, each prefixed with a five-byte sequence header. After all chunks have been transmitted, the sender opens a TCP feedback channel and conducts up to two retransmission rounds, sending only the chunks that any receiver reports as missing. This hybrid design preserves the efficiency benefit of single-transmission broadcast for the common case while guaranteeing eventual reliable delivery whenever the loss rate is not pathological.

### 3.6 Configurable Transfer Policy

The system enforces administratively configurable restrictions on every transfer attempt. The configuration module maintains a small key-value store in a dedicated SQLite database that holds two principal entries: a maximum permitted file size, defaulting to one hundred megabytes, and a list of blocked file-name extensions. Before any encryption or transmission begins, the validation routine consults this store and rejects any file whose size exceeds the limit or whose extension appears in the blocked list. Storage of the policy in a database allows an administrator to update the restrictions at runtime without requiring a code change or application restart.

### 3.7 Resumable Transfer State

For large transfers and intermittent network conditions, the system maintains a persistent record of per-chunk delivery progress in a dedicated transfer-state database. Each transfer is identified by a sixteen-character hexadecimal session token derived from the file name, the SHA-256 digest of the file content and the peer IP address; this identifier is stable across application restarts. As each chunk is transmitted and acknowledged, the corresponding index is appended to a JSON array stored against the session token. If a transfer is interrupted, the next invocation reads back the array and resumes from the first unacknowledged chunk, eliminating the need to retransmit data that has already been delivered.

## 4. SYSTEM ARCHITECTURE AND DESIGN

### 4.1 Architectural Layers

The system is partitioned into five horizontal layers that interact through narrow, well-defined interfaces. The user-interface layer is responsible only for capturing user intent and rendering status to the operator; it does not perform any cryptographic work directly. The authentication-and-authorisation layer is invoked once per user session and once per administrative action. The transfer layer encapsulates all socket programming and chunking logic. The cryptography layer is invoked by the transfer layer and never by the user interface, ensuring that key material does not traverse interface code. The persistence layer is invoked by the authentication, audit and configuration modules.

### 4.2 Module Breakdown

The cryptographic engine, implemented in the encryptor.py module, exposes generation and verification routines for AES keys, RSA keypairs and SHA-256 digests. The unicast sender and receiver pair realise the per-session ephemeral-key TCP protocol. The broadcast sender and receiver pair realise the UDP delivery and TCP feedback retransmission scheme. The authentication module auth.py, the audit module audit.py, the configuration module config.py and the transfer-state module transfer_state.py each manage one of the four SQLite databases. The desktop application is implemented in secure_transfer_gui.py using PyQt5, and the web dashboard is implemented in web_dashboard.py using Flask.

### 4.3 Data Flow

A typical unicast transfer proceeds as follows. The operator authenticates through either the desktop or the web interface; the authentication module verifies the supplied password against the stored bcrypt hash and grants the requested session. The operator selects a file and a receiver address. The configuration module validates the file against the size limit and extension blacklist. The transfer module establishes a TCP connection to the receiver. The receiver generates an ephemeral RSA keypair and sends the public key to the sender. The cryptographic engine generates a fresh AES key, encrypts the file and produces the encrypted payload. The transfer module wraps the AES key under the receiver's public key and transmits both the wrapped key and the encrypted payload in chunks of 64 KB. The receiver unwraps the AES key, decrypts the payload, verifies the SHA-256 digest and returns an acknowledgement. Both the sender and the receiver write audit entries.

### 4.4 Database Design

The four SQLite databases are kept distinct so that each may evolve independently. The user database carries authentication and role data; the audit database carries the immutable transfer history; the configuration database holds two key-value entries; and the transfer-state database carries the chunk-level progress records that allow interrupted transfers to be resumed. All four are opened in WAL journal mode so that read traffic from the web dashboard does not block write traffic from the desktop application.

### 4.5 Threat Model

The system is designed against a network adversary who is able to passively observe all traffic on the shared local segment and who is able to inject, modify or drop packets at will. The adversary is not assumed to control either endpoint of the transfer, nor to possess any pre-existing cryptographic material. Within this threat model, the system provides three guarantees. Confidentiality is preserved by ensuring that the per-session AES key never appears on the wire in plaintext and that the file content is transmitted only in CBC-encrypted form. Integrity is preserved by the SHA-256 digest embedded in every encrypted payload. Authentication and authorisation are preserved by storing only bcrypt-hashed credentials, by enforcing role checks at every administrative entry point and by using parameterised SQL placeholders throughout the data-access modules to eliminate injection vectors.

## 5. IMPLEMENTATION

### 5.1 Technology Stack

The complete system is implemented in Python 3.11. Cryptographic operations rely on PyCryptodome (≥ 3.20), which provides well-audited implementations of AES-CBC, RSA-OAEP, SHA-256 and secure random-byte generation. Password hashing is delegated to the bcrypt package (≥ 4.0). The desktop interface is built with PyQt5 (≥ 5.15), the web interface with Flask (≥ 3.0). Persistence uses the standard library sqlite3 module with WAL mode enabled at connection creation.

### 5.2 Encryption Module

The encryption module exposes a small public interface. The generate_key() function returns a cryptographically random 32-byte sequence suitable for use as an AES-256 key. The encrypt_file(input, output, key) function reads the input file, computes a SHA-256 digest of its contents, generates a fresh 16-byte initialisation vector, applies AES-256-CBC encryption with PKCS#7 padding and writes the concatenation of IV, digest and ciphertext to the output file. The decrypt_file(input, output, key) function reverses the operation and returns a boolean indicating whether the recomputed SHA-256 digest matches the embedded one. The RSA helpers generate_rsa_keypair(), rsa_encrypt_key() and rsa_decrypt_key() wrap and unwrap an AES key using the OAEP scheme.

### 5.3 Authentication Module

The authentication module exposes registration, login and role-management routines. The register() function validates the requested username and password, computes a fresh bcrypt hash with a new salt and inserts the row into the users table. The login() function retrieves the stored hash for the requested username, refuses to authenticate users whose role is blocked, and applies bcrypt.checkpw to verify the supplied password against the stored hash. The role-management routines update_user_role() and toggle_user_block() enforce the safeguards described in Section 3.3.

### 5.4 Transfer Modules

The unicast sender opens a TCP connection to the configured receiver address, reads the receiver's ephemeral RSA public key from the first frame of the response, wraps the freshly generated AES key under that public key and transmits the wrapped key, the file name and the encrypted payload in length-prefixed frames. The unicast receiver performs the reverse operation. Both sides timeout after thirty seconds of inactivity to avoid leaking sockets on a failed connection.

The broadcast sender constructs a metadata header containing the wrapped AES key, the file name, the chunk count and the total payload size in bytes. The header is transmitted by UDP broadcast, followed by the indexed chunks at a controlled inter-chunk interval of five milliseconds to avoid receiver overrun. After the final chunk has been sent, the sender listens on a TCP port for retransmission requests and serves at most two retransmission rounds.

### 5.5 User Interfaces

The desktop application presents five tabs to the authenticated user: Unicast, Broadcast, History, Users and Settings. The Users and Settings tabs are visible only to administrators. Each transfer tab provides a file picker, an address field, a Send button and a scrollable console log. The web dashboard exposes five pages: a home overview with summary statistics, a transfers table that mirrors the audit log, a settings page for policy administration, a users page for account management and a security page summarising the active cryptographic configuration.

### 5.6 Algorithm for the Unicast Send Operation

The unicast send operation proceeds as the following sequence of steps given an authenticated sender, a target receiver address and a chosen file path:

1. Validate the file against the configured size and extension restrictions; abort on failure and record an audit entry with status failure.
2. Generate a fresh 32-byte AES key from the operating system's cryptographic random pool.
3. Generate a fresh 16-byte initialisation vector from the same source.
4. Compute the SHA-256 digest of the original plaintext file content.
5. Encrypt the file under AES-256-CBC; write the concatenation of IV, digest and ciphertext to a temporary file.
6. Open a TCP connection to the receiver address with a 30-second inactivity timeout.
7. Receive the receiver's ephemeral RSA-2048 public key as a length-prefixed frame.
8. Wrap the AES key under the received public key using PKCS#1 v2.1 OAEP padding.
9. Transmit the file name, the wrapped AES key and the total payload size as length-prefixed frames.
10. Transmit the encrypted payload in successive 64-kilobyte chunks, updating the on-screen progress indicator after each chunk.
11. Await a three-byte acknowledgement from the receiver; ACK indicates successful integrity verification, ERR indicates failure.
12. Record the outcome in the audit log with the appropriate status code.
13. Delete the temporary encrypted file regardless of success or failure.

The receiver's counterpart procedure generates and transmits the ephemeral RSA keypair, receives and unwraps the AES key, receives and reassembles the ciphertext, decrypts it under the recovered AES key, recomputes the SHA-256 digest and returns the corresponding acknowledgement.

## 6. RESULTS AND ANALYSIS

### 6.1 Functional Testing

Functional verification was carried out against a set of test files ranging from one kilobyte to one hundred megabytes in size and covering five content categories: plain text, image, document, archive and video. Every test transfer was checked for bit-perfect delivery by computing the SHA-256 digest of the source and received files independently. Across all combinations of file size, content category and transfer mode, the received file matched the source. The user-management subsystem was verified through a suite of unit tests covering registration, login with correct and incorrect passwords, role updates and the administrator-safeguard behaviour of the block routine.

### 6.2 Performance Analysis

Performance measurements were collected over the local loopback interface to isolate the contribution of cryptographic operations from network variability. The wall-clock times for encryption, transmission and acknowledgement were recorded for files of one kilobyte, one megabyte, five megabytes, twenty megabytes, fifty megabytes and one hundred megabytes. Table 6.1 summarises the results.

| File Size | Encrypt (s) | Transfer (s) | Total (s) | Throughput |
|---|---|---|---|---|
| 1 KB | < 0.01 | < 0.01 | 0.03 | — |
| 1 MB | 0.02 | 0.04 | 0.06 | 16.7 MB/s |
| 5 MB | 0.08 | 0.18 | 0.26 | 19.2 MB/s |
| 20 MB | 0.31 | 0.72 | 1.03 | 19.4 MB/s |
| 50 MB | 0.77 | 1.83 | 2.60 | 19.2 MB/s |
| 100 MB | 1.55 | 3.66 | 5.21 | 19.2 MB/s |

*Table 6.1: Unicast transfer performance over the loopback interface.*

The data show that throughput stabilises at approximately 19 MB/s for files larger than one megabyte. AES-256-CBC accounts for roughly thirty per cent of the total elapsed time; socket input and output account for the remainder. The relationship between file size and total time is linear, indicating no quadratic overhead in chunking or in the integrity-verification path.

### 6.3 Security Analysis

Three classes of attack were simulated against the deployed system. In the first, a packet-capture tool running on a second machine on the same LAN recorded the complete TCP exchange of an active transfer. Manual inspection of the captured stream confirmed that no plaintext AES key was visible: the wrapped key appears as a 256-byte high-entropy OAEP ciphertext, and the file payload appears as undifferentiated ciphertext. In the second test, a single byte of the encrypted payload was modified in transit by means of a man-in-the-middle interception script. The receiver detected the corruption through SHA-256 mismatch, refused to write the decrypted file and recorded the failure in the audit log. In the third test, the login routine was exercised with classical SQL injection payloads. Because the underlying query uses parameterised placeholders, the malicious input was treated as a literal username string and no row was returned.

### 6.4 Broadcast Transfer Performance

The broadcast mode was evaluated using two receivers on the same local segment. A twenty-megabyte file was distributed without any artificial loss; both receivers reported successful delivery in 2.8 seconds with no retransmission rounds invoked.

The same transfer was then repeated with five chunks dropped at random; the retransmission round delivered the missing chunks in an additional 0.4 seconds, representing an overhead of approximately fourteen per cent. Three concurrent twenty-megabyte unicast transfers from a single sender completed in 3.4 seconds in total, indicating that per-transfer overhead scales approximately linearly with the number of receivers.

### 6.5 Resumable Transfer Verification

To verify the resumable-transfer logic, a fifty-megabyte file transmission was deliberately interrupted by terminating the sender process after the progress indicator passed forty per cent. The transfer-state database recorded twenty completed chunks against a total of fifty. Restarting the sender produced the same session token from the file name, content digest and peer address combination, allowing the sender to identify the previous incomplete session and resume transmission from chunk index twenty-one. The final file received by the receiver matched the source file byte-for-byte, confirming that the resume mechanism preserves end-to-end integrity.

### 6.6 Comparative Analysis

Table 6.2 summarises the operational characteristics of the proposed system relative to three widely deployed alternatives. The comparison illustrates the gap that the present work fills.

| Property | OpenSSH SFTP | FileZilla | WinSCP | Proposed SFTS |
|---|---|---|---|---|
| Inspectable source | partial | partial | partial | full |
| Desktop GUI | no | yes | yes | yes |
| Web dashboard | no | no | no | yes |
| Broadcast support | no | no | no | yes |
| Embedded audit log | no | no | partial | yes |
| File-type policy | no | no | no | yes |
| Resumable transfers | partial | yes | yes | yes |
| Implementation | C | C++ | C++ | Python |

*Table 6.2: Operational comparison with established tools.*

## 7. CONCLUSION AND FUTURE SCOPE

### 7.1 Conclusion

This paper has presented the design, implementation and evaluation of a Secure File Transfer System implemented in Python. The implementation combines symmetric encryption under AES-256-CBC with asymmetric session-key wrapping under RSA-2048-OAEP, augments these with SHA-256 payload integrity and bcrypt-based credential storage, and records every transfer in a structured persistent log. The resulting application supports both unicast and broadcast delivery and is controlled through either a desktop client or a web dashboard. Functional testing confirms bit-perfect delivery across a wide range of file sizes and types; performance measurements demonstrate sustained throughput of approximately 19 MB/s on a local network; and security testing against passive eavesdropping, ciphertext tampering and credential-bypass attempts confirms that the cryptographic guarantees hold. The transparent Python implementation makes the system suitable both as an operational tool for small-scale secure exchange and as an instructional artefact for the study of applied cryptography.

### 7.2 Future Scope

A number of directions for extension have been identified. The Flask web dashboard, currently exposed over plain HTTP, should be served over TLS in any production deployment. The current implementation operates only within a LAN; extension to wide-area scenarios would require NAT traversal techniques or the introduction of a relay server. Migration from AES-CBC with an external SHA-256 hash to AES-GCM would unify the confidentiality and integrity steps under a single authenticated-encryption primitive and align the design with the recommendations of TLS 1.3. Migration of the password-hashing scheme from bcrypt to Argon2id would adopt the most recent recommendation of the Password Hashing Competition. Storage of RSA private keys in a hardware security module would protect against host compromise. Finally, integration of SQLCipher would provide transparent at-rest encryption of all four SQLite databases.

## REFERENCES

[1] J. Postel and J. Reynolds, “File Transfer Protocol,” RFC 959, Internet Engineering Task Force, October 1985.

[2] P. Ford-Hutchinson, “Securing FTP with TLS,” RFC 4217, Internet Engineering Task Force, October 2005.

[3] T. Ylonen and C. Lonvick, “The Secure Shell (SSH) Transport Layer Protocol,” RFC 4253, Internet Engineering Task Force, January 2006.

[4] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 7th ed. Pearson Education, 2017.

[5] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 20th Anniversary ed. Wiley, 2015.

[6] RSA Laboratories, *PKCS #1 v2.1: RSA Cryptography Standard*, June 2002.

[7] N. Provos and D. Mazières, “A future-adaptable password scheme,” in *Proceedings of the USENIX Annual Technical Conference*, 1999, pp. 81–91.

[8] A. Lioy and G. Ramunno, “An overview of audit log architectures for distributed security systems,” *Journal of Network and Systems Management*, vol. 20, no. 2, pp. 187–212, 2012.

[9] R. Prakash and S. Saravanan, “Hybrid AES-RSA encryption for secure file transfer applications,” *International Journal of Computer Applications*, vol. 175, no. 11, pp. 22–28, 2020.

[10] P. Yadav, A. Mishra and S. Kumar, “A comparative evaluation of Python cryptographic libraries for secure application development,” *International Journal of Network Security*, vol. 23, no. 4, pp. 612–620, 2021.

[11] National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, FIPS Publication 197, November 2001.

[12] National Institute of Standards and Technology, *Recommendation for Key Management*, NIST SP 800-57 Part 1 Rev. 5, May 2020.

[13] M. Bellare, A. Desai, E. Jokipii and P. Rogaway, “A concrete security treatment of symmetric encryption,” in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, 1997, pp. 394–403.

[14] D. J. Bernstein, “ChaCha, a variant of Salsa20,” *Workshop Record of SASC*, 2008.

[15] C. Percival and S. Josefsson, "The scrypt Password-Based Key Derivation Function," RFC 7914, Internet Engineering Task Force, August 2016.

[16] A. Biryukov, D. Dinu and D. Khovratovich, "Argon2: New generation of memory-hard functions for password hashing and other applications," in *Proceedings of the IEEE European Symposium on Security and Privacy*, 2016, pp. 292–302.

[17] D. R. Ferraiolo and D. R. Kuhn, "Role-Based Access Controls," in *Proceedings of the 15th National Computer Security Conference*, 1992, pp. 554–563.

[18] D. Gupta and N. Sharma, "Performance analysis of SQLite in concurrent read-write workloads with WAL journaling," *International Journal of Database Theory and Application*, vol. 12, no. 3, pp. 11–22, 2019.

[19] S. Verma and R. Tiwari, "Reliable UDP broadcast for chunked file delivery," *International Journal of Computer Networks and Communications*, vol. 14, no. 5, pp. 71–84, 2022.

[20] J. Chen and R. Venkatesan, "Performance and security comparison of AES-CBC and AES-GCM for file-transfer applications," *International Journal of Network Security*, vol. 20, no. 3, pp. 512–521, 2018.

**Author Profiles**

Mrs.K.Pavani Working as Assistant & Head of Department of MCA ,in SRK Institute of technology in Vijayawada. She done with MCA and M. Tech in Computer Science .She has 10 years of Teaching experience in SRK Institute of technology, Enikepadu, Vijayawada,NTR District. Her area of interest includes AI ML, etc**.**

Ms.K.Kavya is an MCA Student in the Department of Computer Application at SRK Institute Of Technology, Enikepadu, Vijayawada, NTR District. She has Completed Degree in B.B.A from Andhra Loyola College Vijayawada. Her area of interest are DBMS and Cyber Security.