International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 13 Issue: 11 | Nov 2026 www.irjet.net p-ISSN: 2395-0072

Secure File Sharing with End-to-End Encryption

Prajakta Vidhatee[1], Sanskruti Ohol[2], Bhavika Kalamkar [3], PROF. S. Chechare[4]

[1,2,3] (Students, Department of Computer Engineering), International Centre of Excellence in Engineering & Management

[4] (Professor, Dept. of Computer Engineering), International Centre Of Excellence in Engineering & Management

---

Abstract - *Individuals and organisations increasingly exchange confidential documents through cloud services. Yet most sharing platforms can read the files they store, so a single server breach or a malicious insider can expose private data. This paper presents the design and implementation of a secure file-sharing system with end-to-end encryption (E2EE), built with Python, the Flask web framework and the Supabase backend platform. The sender encrypts files with AES-256 in Galois/Counter Mode (GCM) and wraps the one-time AES key with the receiver's RSA-2048 public key using OAEP padding, so that only the receiver's private key can recover it. The server and the cloud store therefore hold only ciphertext, wrapped keys and metadata. A SHA-256 digest and the GCM authentication tag provide integrity verification. Flask exposes a REST API with JSON Web Token (JWT) authentication, role and ownership checks, and an audit trail. Supa base supplies PostgreSQL tables protected by row-level security and a private Storage bucket for encrypted blobs. The prototype was tested across registration, key exchange, upload, sharing, download, tamper-detection and access-control scenarios. The work demonstrates a practical, modular approach to combining hybrid cryptography with a lightweight web backend so that confidentiality does not depend on trusting the server.*

Keywords - Secure file sharing, end-to-end encryption, AES-GCM, RSA-OAEP, hybrid cryptography, Flask

1. INTRODUCTION

Cloud storage and online sharing have become the default way to move documents between students, employees and organisations. In the usual model, the provider encrypts data in transit and at rest but holds the keys, so the provider, a compromised administrator account or an attacker who breaches the server can read every file. For sensitive material such as academic records, legal papers, medical reports and business plans, this trust assumption is unacceptable. The end-to-end argument in system design holds that some functions can be implemented completely and correctly only at a communication system's endpoints [1]; confidentiality is one such function.

End-to-end encryption (E2EE) addresses the problem by ensuring that plaintext exists only on the sender's and receiver's devices. Building a usable E2EE file-sharing service, however, requires more than choosing a cypher. The system must safely generate and distribute keys, authenticate users, authorise access to individual files, detect tampering, store large encrypted objects efficiently, and keep an audit record, all without giving the server any ability to decrypt. Many prototypes either encrypt files only on the server, which defeats the purpose, or demonstrate a cipher without a complete application around it.

This paper describes a secure file-sharing system with end-to-end encryption, developed in Python with Flask, AES/RSA hybrid cryptography and Supabase. The aim is to deliver registration, key management, encrypted upload, controlled sharing, verified download and auditing in one integrated, maintainable application. The main objectives are: (i) to encrypt every file on the client with AES-256-GCM; (ii) to protect each file key with the receiver's RSA public key [2], [6]; (iii) to expose the functions through a structured Flask REST API secured by hashed credentials and JWT [7]; (iv) to store ciphertext, metadata and access rules in Supabase with row-level security; and (v) to verify integrity and record every sensitive action.

This work contributes a layered reference architecture for end-to-end encrypted file sharing, a hybrid encryption and key-wrapping pipeline with integrity checking, a storage strategy that keeps the cloud blind to file contents, and a systematic testing and deployment approach. The remainder of the paper is organised as follows: Section 2 reviews related work, Section 3 states the problem and requirements, Section 4 gives the system overview and technology stack, Section 5 presents the architecture, Sections 6 to 8 describe the implementation of the platform, the cryptographic engine and the key and storage strategy, Section 9 covers testing and deployment, Section 10 discusses outcomes and limitations, and Section 11 concludes.

2. RELATED WORK

Symmetric encryption and authenticated modes. The Advanced Encryption Standard (AES), based on the

Rijndael cypher, is the dominant symmetric cypher for bulk data because it is fast and well analysed [3]. Plain block-cypher modes provide confidentiality only. Galois/Counter Mode (GCM) is an authenticated-encryption mode that encrypts data and produces an authentication tag in a single pass, so it detects any ciphertext modification at decryption; it requires a unique initialisation vector (IV) for every encryption under the same key [4].

Public-key cryptography. The RSA algorithm shows that a public key can be published openly while the matching private key remains secret [2]. RSA is too slow for large files and can only encrypt short inputs, so it is used to protect small values such as symmetric keys. Optimal Asymmetric Encryption Padding (OAEP) randomises RSA encryption and makes it resistant to chosen-ciphertext attacks [5], [6], and PKCS #1 specifies its use [5].

Hybrid cryptography and key management. Practical secure systems combine both families: a fresh symmetric key encrypts the data and a public key wraps that key. Private keys must be protected at rest; password-based key derivation functions such as PBKDF2 slow down guessing

attacks against a user's passphrase [9], and general key-management guidance covers key lifetimes, storage and rotation [10].

Web architecture and authentication. Representational State Transfer (REST) provides a stateless client-server style suited to web APIs [8]. JSON Web Tokens give a compact, signed way to carry claims and enable stateless authentication [7]. Backend-as-a-service platforms such as Supabase combine PostgreSQL, authentication, object storage and row-level security in one package, so a small team can build a secure data layer quickly. This work integrates these ideas into a single Python/Flask application with a client-side cryptographic core.

3. PROBLEM STATEMENT AND REQUIREMENTS

Existing small-scale file-sharing tools typically suffer from four gaps: (a) files are stored in readable form or encrypted with keys the server controls, (b) sharing links are unauthenticated or never expire, (c) tampering with stored files goes undetected, and (d) there is no record of who accessed what. The proposed system is therefore specified against the requirements listed in Table 1.

TABLE 1. FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

ID	Requirement	Type
R1	Secure registration, login and role-based access	Functional
R2	Generation and registration of a per-user RSA key pair	Functional
R3	Client-side AES-256-GCM encryption of every file before upload	Functional
R4	Per-receiver sharing through RSA-OAEP key wrapping	Functional
R5	Download with integrity verification and local decryption	Functional
R6	Revocation of shares and deletion of files by the owner	Functional
R7	Audit log of uploads, shares, downloads and failures	Functional
R8	Server and storage never hold plaintext files or private keys	Non-functional
R9	Modular, maintainable and deployable code base	Non-functional

Two principal actors interact with the system. The sender registers, encrypts and uploads files and chooses who may read them. The receiver authenticates, retrieves shared files and decrypts them locally. A third role, the administrator, manages accounts and reviews audit records but cannot read file contents. The server acts as an untrusted but available relay and key directory.

Threat model. The system assumes an honest-but-curious or compromised server, database and storage provider, and an attacker who can observe or modify network traffic and stored objects. It does not defend against malware on a user's own device or against a user who discloses their private key or passphrase.

4. SYSTEM OVERVIEW AND TECHNOLOGY STACK

The design rationale can be summarised by three questions. Why: to give users a simple way to share confidential files without having to trust the service that carries them. How: by combining AES-256-GCM, RSA-

OAEP, Python, Flask and Supabase with JWT authentication, row-level security and integrity checks. What: a secure, integrated file-sharing application that provides key management, encrypted storage, controlled sharing and auditing. Table 2 lists the technology stack and the role of each component.

TABLE 2. TECHNOLOGY STACK AND ROLE OF EACH COMPONENT

Layer	Technology	Role
Client	Python (cryptography library), HTML, CSS, JavaScript	Key generation, file encryption and decryption, upload and download
Backend	Python, Flask	REST APIs, request validation, middleware, services
Database	Supabase PostgreSQL	Users, public keys, file metadata, shares, audit logs
Object store	Supabase Storage (private bucket)	Encrypted file blobs and signed download URLs
Cryptography	AES-256-GCM, RSA-2048 OAEP, SHA-256, PBKDF2	Confidentiality, key wrapping, integrity, passphrase protection
Security	JWT, password hashing, row-level security	Authentication, authorisation and data isolation
Tooling	VS Code, Git/GitHub, Postman, pytest	Development, collaboration, API and unit testing

Python was chosen because its cryptography libraries expose well-reviewed AES-GCM and RSA primitives; Flask keeps the API layer thin and modular; Supabase provides PostgreSQL, authentication and storage without custom infrastructure; and PostgreSQL row-level security enforces per-user data isolation inside the database itself.

Development methodology. The project was carried out in staged increments, each producing a working result that the next stage builds on: (1) studying the cryptographic foundations; (2) setting up the Python, Flask and Supabase environment; (3) designing the database schema and storage bucket; (4) creating the Flask REST API; (5) implementing authentication and authorisation; (6) building the key-generation and key-registration module;

(7) implementing AES-GCM file encryption and RSA key wrapping; (8) developing sharing, download and decryption; (9) adding integrity checks and audit logging;

(10) testing; (11) deployment; and (12) preparing documentation, including UML diagrams, test evidence and presentation material. This ordering keeps the platform usable at every stage and allows the security features to be added without disturbing the core flows.

5. SYSTEM ARCHITECTURE

The system follows a layered client-server architecture (Fig. 1). The client layer performs all cryptographic operations, so plaintext and private keys never leave the user's device. It communicates over HTTPS with an application layer built with Flask, which groups logic into authentication, key, file, share and audit services. The data layer combines Supabase PostgreSQL as the system of record for metadata with Supabase Storage as the container for encrypted blobs.

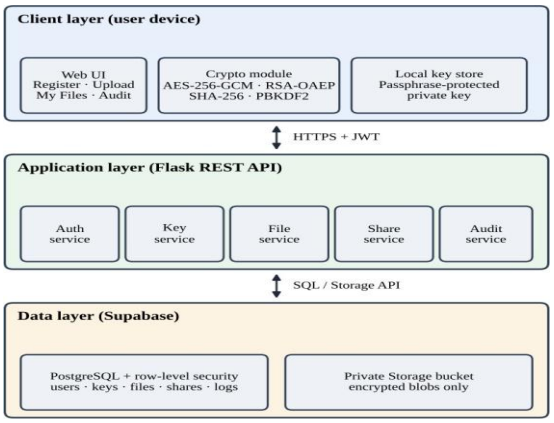


Fig. 1. Layered architecture of the Secure File Sharing system

The codebase is separated into client, backend and configuration directories to clarify responsibilities and simplify maintenance. In the backend, routes, controllers, middleware and services are kept apart so that each can be tested and changed independently. Table 3 lists the principal database tables and the information they hold.

TABLE 3. PRINCIPAL DATA TABLES

Table	Contents
users	ID, name, email, hashed password or Supabase Auth ID, role (user/admin)
public_keys	User ID, RSA public key (PEM), key fingerprint, creation time. Never the private key
files	File ID, owner, original name, size, IV, SHA-256 of ciphertext, storage path
shares	File ID, receiver ID, RSA-wrapped AES key, permissions, expiry, revoked flag
audit_logs	User, action (upload, share, download, revoke, failure), timestamp, client address
Storage bucket	Encrypted file blobs only, in a private bucket

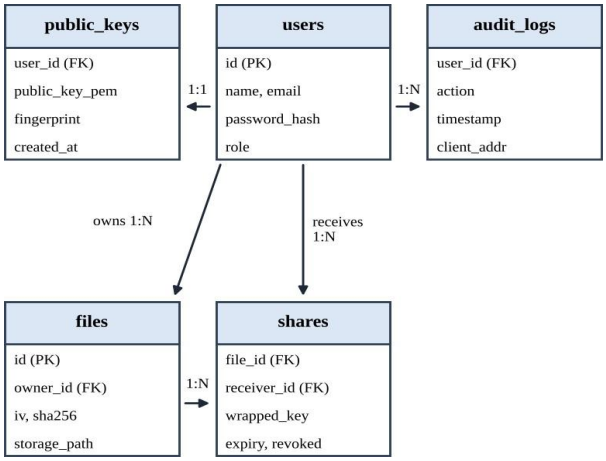


Fig. 2. Entity relationships among the principal tables

As Fig. 2 shows, a user owns files, holds one registered public key, receives shares and generates audit records. The share record is the only place where a file key appears, and

it appears only in wrapped form, which keeps the access-control logic independent of the file contents.

6. IMPLEMENTATION OF THE CORE PLATFORM

6.1 Foundation and environment setup

Development began with the foundations: Python, the cryptography and requests libraries, Flask, Git and GitHub, followed by REST design, JSON handling and secure coding practices. The environment comprises Python virtual environments for dependency isolation, Visual Studio Code for editing and debugging, and a Supabase project whose database connectivity, Storage bucket and API keys were validated before use.

6.2 Client application

The client is a Python module with a simple web interface. It provides Register, Login, Upload, My Files, Shared With Me and Audit screens. Cryptographic work is isolated in one module so that the same code serves the command-line and web clients. Table 4 lists the main screens and their purpose.

TABLE 4. MAIN USER INTERFACE SCREENS

Screen	Purpose
Register / Login	Validated forms; key-pair generation on first registration
Upload	File picker, receiver selection, local encryption progress
My Files	Files owned, share list, revoke and delete actions
Shared With Me	Files shared by others, integrity status, download and decrypt
Audit	Personal activity history; full history for administrators
Admin	Account management and audit review without file access

6.3 Backend and REST APIs

The server is configured with Python and Flask and structured into blueprints, controllers, middleware and services. REST endpoints are provided for users, keys, files, shares and audit logs. Reliability is strengthened through input validation, centralised error handling, structured logging and secure configuration of secrets, including the Supabase service-role key, which is used only on the server.

6.4 User authentication and authorisation

Registration validates submitted details, enforces required fields and prevents duplicate accounts. Passwords are hashed with a strong one-way, salted algorithm before storage, so credentials stay protected even if database records are exposed. At login, the system verifies the submitted password, and a signed JWT access token with a short lifetime is issued [7]. Middleware protects private routes by checking the token and separates user and administrator permissions, and ownership checks ensure that only a file's owner can share, revoke or delete it.

6.5 Security measures

Security is treated as a cross-cutting concern. Passwords are never stored in plain text. Tokens are signed and verified on every protected request. Input validation at the API boundary rejects malformed data early, and centralised error handling avoids leaking internal details. Uploads are limited in size and type, and client-supplied metadata is not trusted. Configuration values such as database URLs, JWT secrets and API keys are kept in environment variables rather than in source code, with least-privilege access and periodic rotation, and CORS and HTTPS-only policies restrict how the API is reached in production.

6.6 REST API design

Table 5 lists the main resource groups exposed by the Flask API. Protected routes pass through authentication middleware; administrative routes pass through an additional role check.

TABLE 5. MAIN API RESOURCE GROUPS

Endpoint	Function	Access
/auth	Register, login, refresh token	Public / user
/keys	Upload own public key, fetch a user's public key	User
/files	Upload ciphertext, list, delete, fetch metadata	User (owner)
/shares	Create share, list shares, revoke, fetch wrapped key	User
/download	Obtain short-lived signed URL for ciphertext	User (receiver)
/audit	List audit entries	User / admin

Every endpoint validates its input, returns consistent JSON and delegates failures to a centralised error handler, so clients receive predictable status codes and messages.

7. CRYPTOGRAPHIC ENGINE

The cryptographic subsystem turns a plaintext file into an encrypted, integrity-protected object that only the intended receiver can open. Its stages are shown in Fig. 3 and described below.

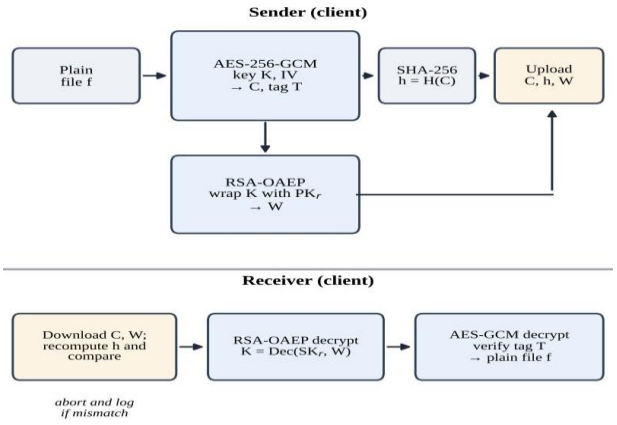


Fig. 3. Hybrid encryption, sharing and decryption pipeline

7.1 Key pairs and private-key protection

At registration, the client generates an RSA-2048 key pair. The client uploads the public key to the key service together with a fingerprint, which users can compare out of band to detect substitution of keys by a malicious server. The private key is never sent to the server. It is stored locally, encrypted with a key derived from the user's passphrase using PBKDF2 with a random salt and a high iteration count [9], so that theft of the key file alone does not reveal the key.

7.2 Authenticated file encryption

For every upload, the client draws a fresh 256-bit AES key and a fresh 96-bit random IV from the operating-system random generator. The file is encrypted with AES-256-GCM [3], [4], which yields ciphertext and a 128-bit authentication tag. Because a new key is generated per file, the IV uniqueness requirement of GCM is met by construction. The file name and size are included as associated data so that renaming or substitution attacks fail verification.

7.3 Key wrapping with RSA-OAEP

The AES key is encrypted with the receiver's RSA public key using OAEP padding with SHA-256 [5], [6]. The result, the wrapped key, is small enough for RSA and can be created separately for each receiver, so sharing one file with several people requires wrapping the same file key several times but encrypting the file only once. Revocation removes the receiver's share record, and for stronger guarantees the owner can re-encrypt the file with a new key.

7.4 Integrity verification

A SHA-256 digest of the ciphertext is stored with the file metadata. On download, the client recomputes the digest before decryption, and AES-GCM then verifies the authentication tag. Any bit flipped in storage or in transit causes decryption to fail with an explicit error, and the failure is written to the audit log instead of corrupted data being returned.

7.5 Encryption and decryption algorithm

Algorithm 1 summarises the sender-side and receiver-side procedures.

Algorithm 1. Share(f, r) and Open(id)

1. Sender authenticates and fetches receiver r's RSA public key PKr and fingerprint from the key service.
2. Generate random AES-256 key K and 96-bit IV; compute C, T ← AES-GCM-Encrypt(K, IV, f, associated data).
3. Compute h ← SHA-256(C); compute W ← RSA-OAEP-Encrypt(PKr, K).
4. Upload C to Supabase Storage; insert file metadata (IV, h, path) and share record (W, receiver, expiry).
5. Receiver authenticates; the API checks that a valid, unrevoked share exists and returns metadata, W and a short-lived signed URL.
6. Receiver downloads C; recomputes SHA-256 and compares with h; aborts and logs a failure if different.
7. Receiver unlocks the local private key, computes K ← RSA-OAEP-Decrypt(SKr, W).
8. Decrypt C with AES-GCM using K, IV and the tag; if tag verification fails, discard the output; otherwise save plaintext f locally.

Steps 1 to 4 run once per file and receiver, while steps 5 to 8 run on each download. File size affects only the AES stages, which are fast and streamable, whereas the RSA operations act on a 32-byte key regardless of file size.

7.6 Delivery and sharing flow

Fig. 4 shows the end-to-end exchange between the sender, the Flask API, Supabase and the receiver. The server only relays ciphertext and wrapped keys; at no point does it receive a plaintext file, a file key in the clear or a private key.

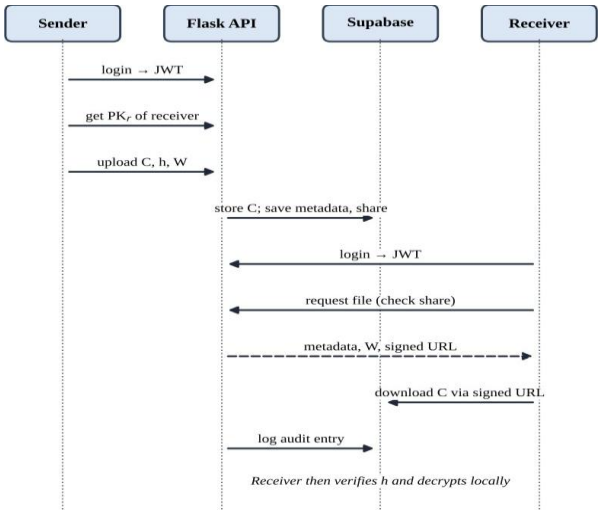


Fig. 4. Upload, share and download sequence

8. KEY DIRECTORY AND SUPABASE STORAGE STRATEGY

The storage strategy has five elements: (1) encrypted blobs are placed in a private Supabase Storage bucket with no public read access; (2) all downloads use short-lived signed URLs that are issued only after the API has verified a valid share; (3) PostgreSQL row-level security policies restrict each row so that owners see their files, receivers see only shares addressed to them, and public keys are readable by authenticated users only; (4) metadata and share records are written in a single transaction so a file never exists without its access record; and (5) deletion and revocation remove the share rows and storage objects and are recorded in the audit log. Table 6 summarises how each category of data is protected.

TABLE 6. DATA PROTECTION POLICY BY DATA CATEGORY

Data	Where held	Protection
File contents	Storage bucket	AES-256-GCM ciphertext, private bucket, signed URLs
File key	shares table	Only RSA-OAEP wrapped for each receiver
Private key	User device only	PBKDF2-derived key encryption, never transmitted
Public key	public_keys table	Readable by authenticated users, fingerprint verifiable
Passwords	users / Auth	Salted one-way hash
Audit records	audit_logs table	Append-only for users; read by owner or admin

Signed-URL lifetimes are configuration parameters because suitable values depend on file size and network speed. If the storage service is unavailable, uploads and downloads fail safely without ever falling back to an unencrypted path.

Scalability considerations

The architecture scales along three axes. The stateless API (JWT rather than server-side sessions) can be replicated behind a load balancer. PostgreSQL can be scaled with indexing on owner, receiver and file identifiers.

Large files can be encrypted in chunks, each with its own derived nonce, so that memory use stays constant and downloads can be streamed.

9. TESTING AND DEPLOYMENT

9.1 Application testing

Testing follows a four-stage cycle of planning, execution, analysis and optimisation. Test cases were defined to cover API behaviour, authentication flows, cryptographic correctness, sharing and revocation, validation rules, error paths and expected user outcomes. Postman was used to run API requests and authentication sequences, and pytest unit tests exercised the encryption, key-wrapping and integrity functions, including known-answer and round-trip tests. Results were reviewed to identify recurring failures and refine assertions, after which defects were diagnosed and the tests repeated. Table 7 summarises the coverage.

TABLE 7. TEST COVERAGE

Area	Tests performed
Authentication	Registration validation, hashed storage, login, JWT checks, role permissions
Cryptography	AES-GCM round trip, unique IV per file, RSA-OAEP wrap and unwrap, wrong-key failure
Sharing	Share creation, receiver-only access, expiry, revocation, multi-receiver sharing

Integrity	Detection of modified ciphertext, wrong tag and swapped metadata
Access control	Row-level security, ownership checks, blocked access to other users' files
Robustness	Input validation, error paths, interrupted transfers, oversized files

9.2 Deployment

The Flask application is deployed behind a production WSGI server with HTTPS enforced and runtime settings configured, and communication with Supabase is confirmed. The Supabase project is configured with production tables, row-level security policies, a private Storage bucket and restricted network access, and persistence and recovery are tested. Database URLs, JWT secrets and service-role keys are held as protected environment variables with least-privilege access and secret rotation. Finally, domains, CORS policies, logging, monitoring and health checks are validated before launch.

10. RESULTS AND DISCUSSION

The implemented system delivers the complete secure-sharing journey, from registration and key generation through encrypted upload, sharing, verified download and local decryption, together with an audit trail. Functional testing confirmed that protected routes accept only valid tokens, that users can read only files shared with them, that the stored objects are unreadable without the receiver's private key, and that modified ciphertext is rejected. Table 8 compares the proposed system with a basic file-sharing portal.

TABLE 8. COMPARISON WITH A BASIC FILE-SHARING PORTAL

Aspect	Basic portal	Proposed system
Encryption	None, or server-side keys	Client-side AES-256-GCM, end to end
Key handling	Server holds keys	Private keys remain on user device
Sharing	Public or permanent links	Per-receiver wrapped keys, expiry, revocation
Integrity	Not checked	SHA-256 plus GCM authentication tag
Server compromise	Files exposed	Only ciphertext exposed
Auditing	Limited or absent	Append-only audit log

From an engineering viewpoint, three design choices were the most significant. First, performing every cryptographic operation on the client removed the need to trust the server for confidentiality. Second, the hybrid design encrypts each file once with AES and wraps only a small key per receiver, which keeps sharing cheap. Third, delegating data isolation to Supabase row-level security places access control next to the data, so a mistake in one API route cannot by itself expose another user's records.

Design trade-offs. Several choices balance competing goals. Because the server never holds private keys, a user who forgets the passphrase and has no backup cannot recover their files. Server-side search and previews are impossible because the server cannot read content. Stateless JWT authentication simplifies scaling, but a token cannot be withdrawn before expiry without extra measures such as short lifetimes or a deny-list. Finally, revoking a share prevents future downloads but cannot take back a file that the receiver has already decrypted and saved.

Key authenticity and privacy. A malicious server could substitute its own public key for a receiver's key and read files sent afterwards. Displaying key fingerprints and encouraging out-of-band verification reduces this risk. Metadata such as file names, sizes, timestamps and the sender-receiver relationship remains visible to the server, so only the minimum metadata should be stored and access to audit records should be limited to the owner and administrators.

Proposed evaluation protocol. To evaluate the system quantitatively, the time taken for encryption, wrapping and decryption can be measured for files of increasing size and compared with unencrypted transfer. Security properties are evaluated separately through tamper tests and access-control tests using the metrics in Table 9.

TABLE 9. METRICS FOR FUTURE QUANTITATIVE EVALUATION

Metric	Description
Encryption/decryption throughput	Megabytes per second for AES-GCM at different file sizes
RSA wrap / unwrap time	Cost of key wrapping per receiver
End-to-end upload and download latency	Total time compared with unencrypted transfer
Memory use	Peak memory with whole-file and chunked encryption
Tamper-detection rate	Fraction of modified objects rejected at download
Unauthorised-access rejection rate	Fraction of invalid requests blocked by API and RLS

Limitations. This paper concentrates on design and implementation. The evaluation was functional, but a large-scale performance and security study was not performed. The system relies on the user's device being

free of malware, does not provide forward secrecy for wrapped file keys, and, in the prototype, loads each file as a whole rather than in chunks. Future experiments should follow the protocol above and report the listed metrics.

11. CONCLUSION AND FUTURE SCOPE

This paper presents a Secure File-Sharing system with end-to-end encryption built with Python, Flask, AES/RSA hybrid cryptography and Supabase. The system integrates client-side AES-256-GCM encryption, RSA-OAEP key wrapping, JWT-secured REST APIs, storage secured by row-level security, integrity verification and audit logging in a modular architecture. The result is a maintainable foundation that lets users share confidential files without trusting the server with their contents.

Future work includes chunked streaming encryption for very large files, digital signatures to authenticate senders, elliptic-curve key agreement with forward secrecy, key backup and recovery mechanisms, group sharing with key rotation, time-limited self-destructing files, multi-factor authentication, formal security analysis and performance benchmarking, and a mobile application client.

ACKNOWLEDGMENT

The authors thank the project guide and the department for their guidance and support during the development of this project.

REFERENCES

[1] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," ACM Trans. Computer Systems, vol. 2, no. 4, pp. 277-288, 1984.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," Commun. ACM, vol. 21, no. 2, pp. 120-126, 1978.

[3] J. Daemen and V. Rijmen, The Design of Rijndael: AES - The Advanced Encryption Standard. Berlin, Germany: Springer, 2002.

[4] M. Dworkin, "Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC," NIST Special Publication 800-38D, Nov. 2007.

[5] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #1: RSA cryptography specifications version 2.2," IETF RFC 8017, Nov. 2016.

[6] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," in Advances in Cryptology - EUROCRYPT '94, LNCS vol. 950. Springer, 1995, pp. 92-111.

[7] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF RFC 7519, May 2015.

[8] R. T. Fielding, "Architectural styles and the design of network-based software architectures," PhD dissertation, Univ. California, Irvine, CA, USA, 2000.

[9] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-based cryptography specification version 2.1," IETF RFC 8018, Jan. 2017.

[10] E. Barker, "Recommendation for key management: Part 1 - General," NIST Special Publication 800-57 Part 1, Rev. 5, May 2020.

BIOGRAPHIES

PROF S.CHECHARE

ASSISTANT PROFESSOR (HOD) DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING INTERNATIONAL CENTER OF EXCELLENCE IN ENGGINERING & MANAGEMENT (ENGINEERING

PRAJAKTA VIDHATE

Pursuing (CSE) ICEEM INTERNATIONAL CENTER OF EXCELLENCE IN ENGGINERING & MANAGEMENT (ENGINEERING

SANSKRUTI OHOL

Pursuing (CSE) ICEEM INTERNATIONAL CENTER OF EXCELLENCE IN ENGINEERING & MANAGEMENT (ENGINEERING

BHAVIKA KALAMKAR

Pursuing (CSE) ICEEM INTERNATIONAL CENTER OF EXCELLENCE IN ENGINEERING & MANAGEMENT (ENGINEERING