WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Secure Vault Storage: A Hybrid Cryptographic Approach for Secure File Handling

[1]Purva Raut, [2]Manmeet Singh Sandhu, [3]Mansi Shelke, [4]Kishori Salokhe, [5]Prof. Pallavi Marulkar

Dept. Computer Engineering, Pillai HOC College of Engineering and Technology, Khalapur, HOC Colony Rd, HOC Colony, Taluka, Rasayani, Maharashtra 410207

ABSTRACT:

The Secure Vault Storage system is a web-based secure file management platform that combines both symmetric and asymmetric cryptographic methods. Built with Python's Flask framework and a MySQL database, the system allows secure file upload, storage, encryption, decryption, and sharing. RSA is used for key exchange, while Fernet is used for data encryption. The goal is to provide a reliable and user-friendly system for users to store and manage sensitive files securely. This paper presents the development, structure, and implementation of the system, along with the key modules that ensure its performance and data protection. The implementation also focuses on activity logging, secure key handling, and ease of access, ensuring high performance and strong confidentiality.

Keywords: Cryptography, Secure File Storage, Flask, RSA, Fernet, MySQL, Hybrid Encryption, Key Management

Introduction:

In the digital era, securing files from unauthorized access has become crucial. Traditional cloud storage systems often store data in plain or weakly encrypted formats. Our Secure Vault Storage system addresses this issue using hybrid encryption (RSA + Fernet), allowing files to be securely stored and shared. The system architecture and encryption model provide layered protection against data breaches and internal misuse.

This system is designed with user-friendly interfaces and strict authentication protocols. The primary advantage lies in combining speed (from symmetric encryption) and security (from asymmetric encryption). The backend leverages Flask and cryptographic libraries to implement core logic, while MySQL stores encrypted file metadata and user credentials. Additionally, the system supports activity tracking, session-based access control, and user-specific file handling with custom keys.

Methodology:

The development of the Secure Vault Storage system followed a structured software engineering methodology, focusing on functionality, security, and user accessibility. The following key stages were involved:

1. **Requirement Analysis**
   The project began with a thorough analysis of security concerns in existing cloud-based file storage systems. Issues such as insecure key storage, lack of file access control, and limited encryption methods were identified. Based on this, the system was designed to incorporate hybrid encryption, user role management, and detailed activity logging.
2. **System Design**
   A modular, three-tier architecture was adopted to separate the presentation layer, application logic, and database services.
   - The **client layer** consisted of an intuitive web interface using HTML, CSS, and Flask templating.
   - The **application layer** (Flask) handled encryption logic, user sessions, and file handling routes.
   - The **data layer** (MySQL) stored user credentials, encryption keys, metadata, and logs securely.

3. Technology Stack Selection

The system was built using Python's Flask framework for its simplicity and flexibility.

- **Cryptography**: RSA (asymmetric) and Fernet (symmetric) for hybrid encryption
- **Database**: MySQL for secure structured storage

4. Module Implementation

The system was divided into multiple modules, developed and tested individually:

- **User Management**: Registration, login, session tracking
- **Encryption Module**: Generation of RSA keys, Fernet-based file encryption, secure key exchange

- **File Operations**: Upload, encrypt, decrypt, delete, and share files with access control
- **Logging System**: Logging of all critical user actions (encrypt, share, delete)

5. Testing and Validation

Each module underwent rigorous testing:

- **Unit Testing** for encryption/decryption and database actions
- **Security Testing** including SQL injection prevention and file tampering resistance
- **Performance Testing** to assess encryption speed and server response under load
- **User Testing** to ensure the interface was easy to use for non-technical users

6. Deployment Setup

The system was deployed locally and tested on multiple devices. Environment variables were used to store sensitive credentials, and a .env file was configured for deployment. Proper directory management ensured that temporary files and encrypted content were stored securely.

Existing System:

In today's digital landscape, file storage and sharing platforms are essential for individuals and businesses. Commonly used services like Google Drive, Dropbox, and OneDrive offer cloud-based storage with basic security features. However, these systems often lack transparency in encryption and key management processes. The following drawbacks were identified during our research:

1. Centralized Key Storage

Most existing platforms store encryption keys on their own servers. While data may be encrypted at rest, the provider holds the decryption keys, making it vulnerable to insider threats or server breaches.

2. Limited End-to-End Encryption

Some tools like Dropbox offer encryption in transit and at rest but not end-to-end encryption. This means files are decrypted temporarily on the provider's server before being delivered to the recipient, exposing a potential attack surface.

3. Inadequate File Sharing Controls

Standard cloud platforms provide basic sharing features such as public or private links. However, there is often no provision to limit access by encryption key or to set temporary access with cryptographic restrictions.

4. Lack of Role-Based Access

Enterprise-grade access control is missing in most public file-sharing systems. Admins cannot fully monitor or restrict how shared files are used after initial access is granted.

5. Insufficient Logging and Monitoring

Most services do not offer detailed file-level activity logs, especially for actions like download, decryption, or sharing. This makes it difficult to audit file access or detect misuse.

6. Open Source Alternatives

Some tools like Cryptomator or VeraCrypt offer local encryption before cloud upload. However, these tools often do not support file sharing, web-based access, or remote key management, limiting their usability in collaborative environments.

Drawbacks of Existing Systems:

Despite the widespread use of cloud storage platforms, several limitations exist in the way current systems handle security and access control. Through comparative research, the following key drawbacks were identified:

1. Centralized Encryption Key Management

Most platforms manage encryption keys on their own servers. This introduces a single point of failure—if the server is compromised, all user data becomes vulnerable, even if encrypted.

2. Absence of Hybrid Encryption

Standard cloud storage systems generally use either symmetric or basic encryption methods. These do not provide the layered protection or secure key sharing mechanisms that hybrid encryption (RSA + Fernet) offers.

3. Limited User Access Control

Role-based access is rarely implemented. Files shared with users are often accessible without granular control over who can decrypt, download, or modify them. There is no separation of duties or customizable user permissions.

4. Weak Audit Trails and Logging
Existing systems lack detailed logging features. They typically do not record when a file is encrypted, decrypted, or shared, and by whom. This makes it difficult to trace unauthorized actions or data breaches.

5. No Secure File Sharing Protocols
File sharing usually relies on simple URLs or email links without encryption or access time limits. This exposes files to risks like unauthorized sharing, misuse, or phishing attacks.

6. No Localized or Offline Encryption Support
Cloud-native tools do not allow users to encrypt files independently before upload or decrypt them offline. Users are dependent on the cloud provider's infrastructure for all encryption operations.

7. Inadequate Support for Advanced Security Features
Features like key rotation, password-derived keys (PBKDF2), biometric login, or access via secure tokens are not supported in most standard solutions, limiting their use in high-security environments.

System Components:

The Secure Vault Storage system is divided into several functional components, each responsible for handling specific operations within the platform. The system architecture ensures clear separation of concerns, efficient encryption workflows, and secure file management. The major components are as follows:

1. User Interface (Client Layer)
This is the web-based front end developed using HTML, CSS, and Flask templates. It provides users with interactive screens for file upload, encryption/decryption, viewing shared files, and managing their profile. The UI ensures ease of use while enforcing security-driven workflows.

2. Authentication Module
Handles secure login and registration of users. Passwords are hashed using bcrypt, and sessions are managed to ensure users stay authenticated across requests. The module includes form validation and secure session handling using cookies and server-side checks.

3. Encryption Module
The core security feature of the system. It uses:

- **RSA** (asymmetric encryption) for secure key exchange between users during file sharing
- **Fernet** (symmetric encryption) for encrypting and decrypting file contents
- **PBKDF2** (Password-Based Key Derivation Function 2) to generate strong encryption keys from user passwords

All files are encrypted before storage and decrypted only after proper authentication.

4. File Management Module
This component allows users to perform file operations such as:

- Upload encrypted files
- Decrypt/download stored files
- Delete old or unwanted files

The system ensures each file is stored in encrypted form and can only be accessed with the appropriate cryptographic keys.

5. Key Management System
Manages encryption keys securely. It stores RSA key pairs for each user and handles key generation, exchange, and verification during file encryption and sharing. Keys are stored in encrypted format within the database and never exposed in plaintext.

6. Database Layer (MySQL)
Maintains all user information, file metadata, encrypted keys, and activity logs. Key tables include:

- users: stores hashed credentials and public keys
- files: stores encrypted file paths, file metadata
- logs: tracks user activity such as login, encryption, and sharing events

The database is protected using best practices like query sanitization and limited access privileges.

7. Logging and Monitoring Module
Tracks and records system events. It logs all critical operations like login attempts, file encryption, decryption, sharing actions, and admin activity. These logs help maintain accountability and support forensic analysis in case of data misuse.

Technical Implementation:

- Backend Framework: Flask (Python) – lightweight and suitable for quick RESTful API development
- Frontend: HTML5, CSS3, Bootstrap – used to design a clean and responsive user interface
- Database: MySQL – chosen for its reliability in handling relational data securely
- Security Libraries:
    - bcrypt for password hashing

- cryptography module (Fernet for symmetric encryption, RSA for asymmetric encryption)
- PBKDF2HMAC for key derivation using user credentials

Database Structure:

The Secure Vault Storage system employs a structured MySQL relational database to manage encrypted data, user credentials, and system logs. The design follows normalization principles to enhance performance, ensure data consistency, and enforce security.

1. users Table

- Stores user-specific information such as unique user ID, email, hashed password, and RSA public key.
- Fields: user_id, username, email, password_hash, public_key, created_at

2. files Table

- Maintains metadata of uploaded files, including filename, owner ID, upload timestamp, and encrypted Fernet key.
- Fields: file_id, file_name, owner_id, encrypted_key, upload_time, file_path

3. shared_files Table

- Manages file sharing relationships between users. Includes encrypted keys specific to recipients.
- Fields: share_id, file_id, shared_by, shared_to, shared_key, access_expiry

4. logs Table

- Records user activities for audit purposes. Tracks login attempts, file operations, and errors.
- Fields: log_id, user_id, action, timestamp, status, ip_address

5. sessions Table *(optional enhancement)*

- Maintains session identifiers for logged-in users. Helps manage session expiration and token-based access.

This schema supports efficient encryption workflows, secure key storage, and quick retrieval of user-specific file data with full logging and traceability.

Challenges Faced:

During the development and deployment of the Secure Vault Storage system, several technical and architectural challenges were encountered. Each challenge was met with a focused solution to ensure the application remained robust, secure, and user-friendly.

1. Secure Key Handling

Implementing a hybrid encryption system required proper management of both symmetric (Fernet) and asymmetric (RSA) keys. Ensuring these keys were securely generated, exchanged, and stored without exposure was complex and required careful cryptographic implementation.

2. File Encryption Efficiency

Encrypting large files caused temporary performance delays during upload. This was mitigated by optimizing memory management and using buffered file streams instead of direct encryption in memory.

3. Sharing Encrypted Files

Allowing one user to share an encrypted file with another required generating user-specific encrypted keys. Handling key exchanges and ensuring correct decryption across users required custom logic to pair keys correctly with user identities.

4. Error Handling and Logging

Developing a detailed logging mechanism that recorded all critical events without leaking sensitive data took significant effort. Error messages were customized to help users without exposing system internals.

5. UI Integration with Backend Logic

Integrating Flask's backend logic with the front-end template and maintaining secure session handling during encryption/decryption actions needed iterative testing.

Results:

The Secure Vault Storage system follows a three-layer architecture that separates the user interface, application logic, and database. This structure improves security, maintainability, and performance.

1. Client Layer (User Interface)

The front end is built using HTML, CSS, and Flask templates. Users can register, log in, upload files, and manage encryption/decryption from a simple web interface. All communication uses secure HTTPS.

2. Application Layer (Flask Backend)

This is the system's core, built with Python Flask. It handles:

- User login and sessions
- File encryption using Fernet
- Key exchange using RSA
- File uploads, downloads, and sharing
- Admin features like logs and access control
-

3. Data Layer (MySQL Database)

Stores user data, file metadata, encrypted keys, and activity logs. Tables include users, files, shared_files, and logs. Data is protected using query sanitization and restricted access.

4. Encryption Model

- Files are encrypted using Fernet (symmetric).
- Keys are shared securely using RSA (asymmetric).
- PBKDF2 is used to strengthen password-derived keys.

This layered approach ensures data security from input to storage, while keeping the system modular and scalable.

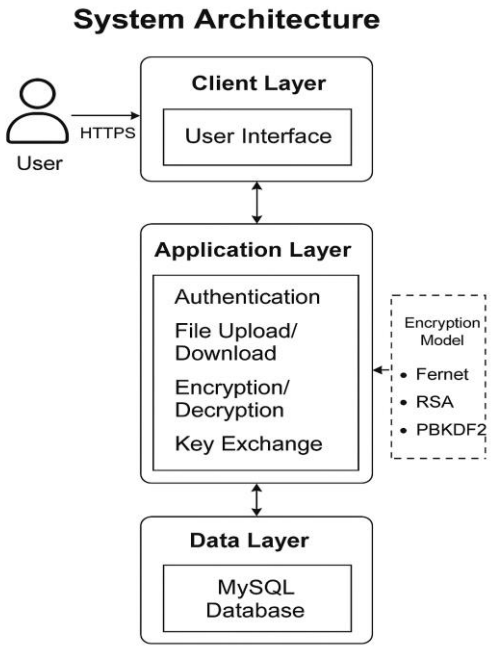

Fig 1: System Architecture

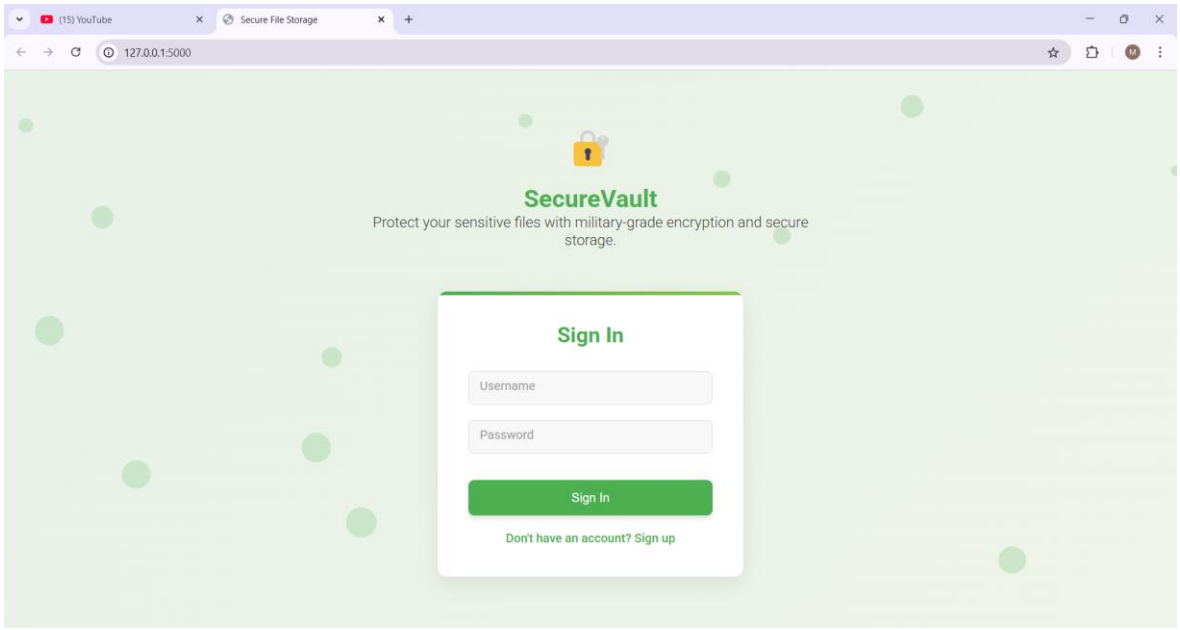

Fig 2: Authentication

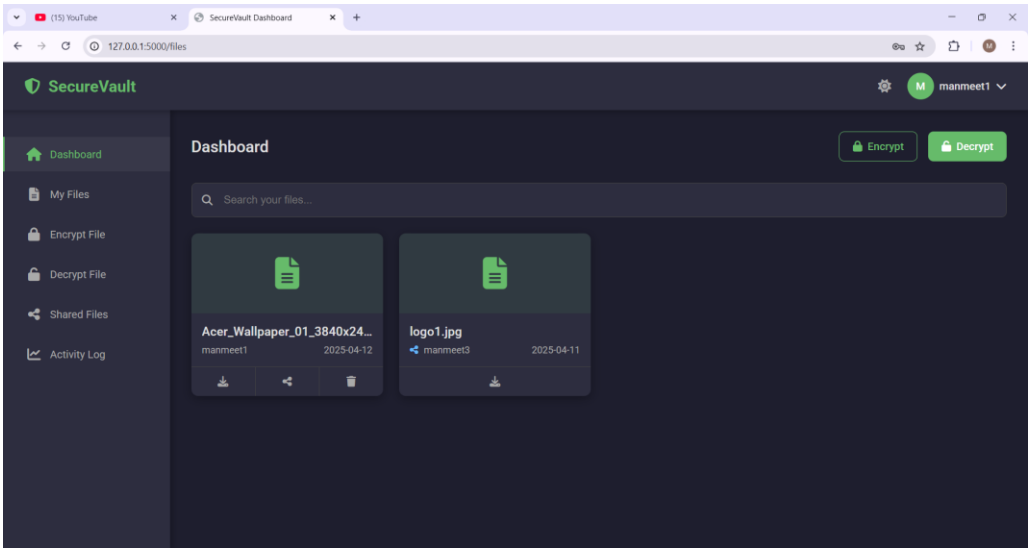

Fig 3: Dashboard

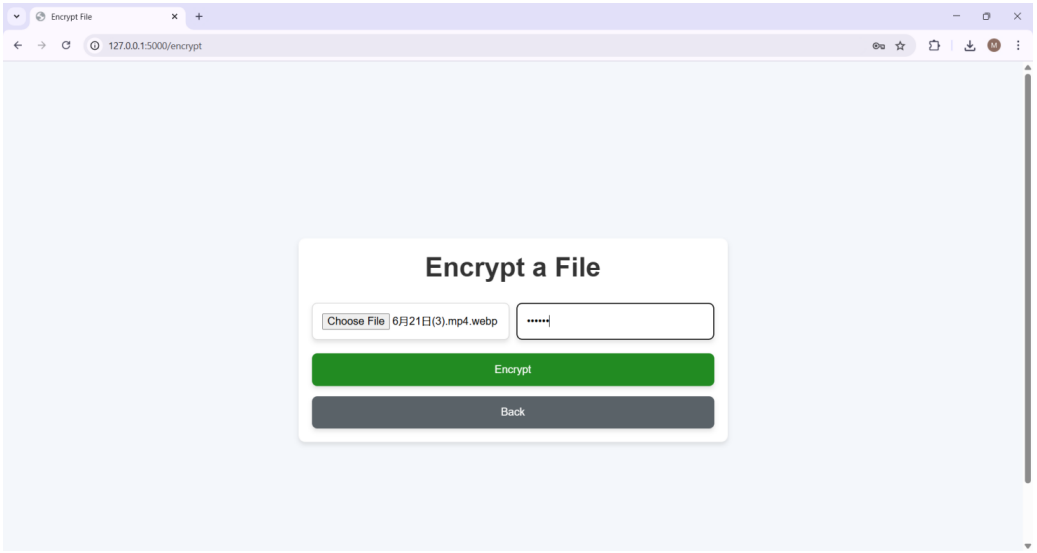

Fig 4: Encrypt a File

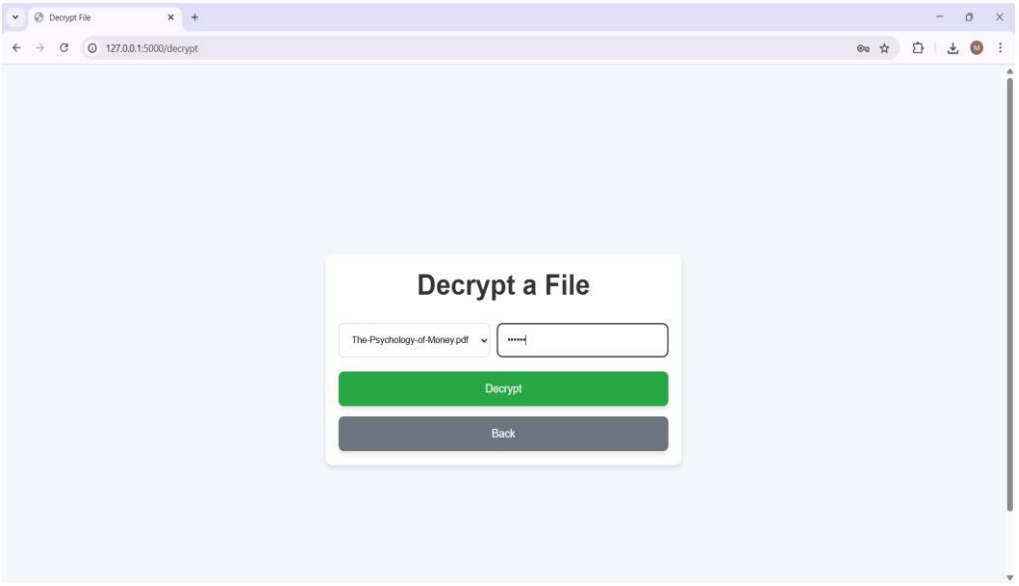

Fig 5: Decrypt a File

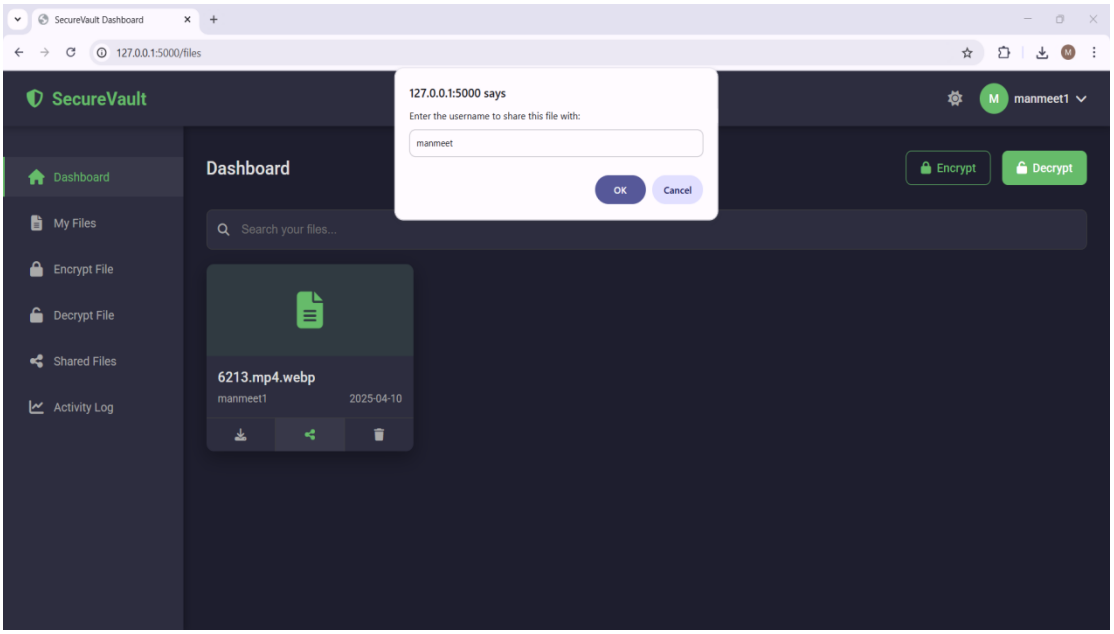

Fig 6: Send a File

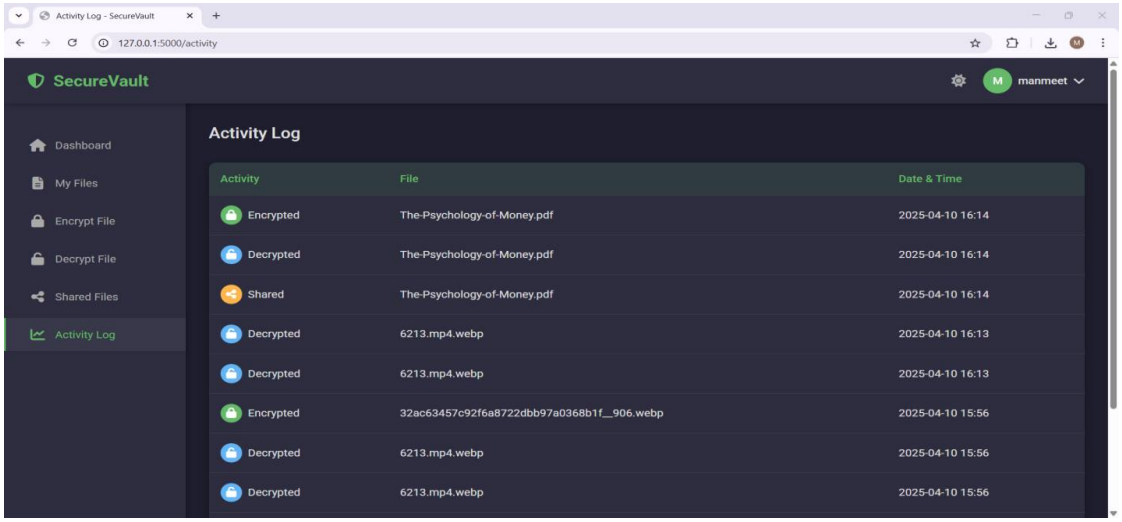

Fig 7: Activity Log

Conclusion

The Secure Vault Storage system provides a practical and secure solution for managing sensitive digital files. By combining symmetric (Fernet) and asymmetric (RSA) encryption methods, the system ensures both performance and high-level data confidentiality.

All file operations—from uploading to sharing—are protected through strong cryptographic techniques and role-based access control. Users can safely store, encrypt, decrypt, and share files with other verified users, while all critical actions are logged for transparency and auditing.

The system addresses key limitations in existing storage platforms by offering secure key management, end-to-end encryption, and detailed logging—all within a user-friendly web interface. With its modular structure and flexible architecture, the application can be extended further to support cloud integration, mobile access, and enterprise-level deployment.

In conclusion, this project demonstrates how modern cryptography can be effectively integrated into file management systems to enhance privacy, usability, and trust in digital environments.

REFERENCES:

List all the material used from various sources for making this project proposal

1. Flask Web Framework Documentation – https://flask.palletsprojects.com
2. MySQL Documentation – https://dev.mysql.com/doc/

3. Python Cryptography Library – https://cryptography.io
4. bcrypt Password Hashing for Python – https://pypi.org/project/bcrypt/
5. RSA Public Key Cryptography Overview – https://www.emc.com/emc-plus/rsa-labs/standards-initiatives/rsa-algorithm.htm