Design and Implementation of a Secure File Encryption and Encrypted Collaboration Platform

Dally
Department of Computer Science and Artificial Intelligence
Rishihood University
Sonipat, India
dally.r23csai@nst.rishihood.edu.in

Udita
Department of Computer Science and Artificial Intelligence
Rishihood University
Sonipat, India
udita.23csai@nst.rishihood.edu.in

Jamathziah
Department of Computer Science and Artificial Intelligence
Rishihood University
Sonipat, India
jamathziah.23csai@nst.rishihood.edu.in

***Abstract*—This paper presents the design and implementation of an integrated secure communication platform consisting of three subsystems: a native C-based file encryption tool with dual interfaces (ncurses TUI and CLI), a real-time encrypted chat application (CipherChat), and a full-stack secure collaboration platform (CipherVault). The native tool leverages POSIX system calls to demonstrate OS fundamentals, implementing Caesar and XOR ciphers with binary-safe I/O. CipherChat provides end-to-end encrypted messaging with Firebase Realtime Database, channel management, and presence tracking. CipherVault implements AES-256-GCM encryption with PBKDF2 key derivation, JWT authentication, role-based access control (RBAC), audit logging, secure file sharing, and room-based collaboration. The system demonstrates practical application of OS concepts including system calls, file descriptors, process lifecycle, concurrency control, and secure storage. Automated testing validates roundtrip encryption integrity for text and binary files, with the native tool achieving 200 MB/s throughput and CipherVault maintaining 15-20% encryption overhead. The platform serves both educational and practical purposes, bridging low-level system programming with high-level web security patterns.**

***Index Terms*—file encryption, AES-256-GCM, PBKDF2, encrypted chat, secure collaboration, POSIX system calls, JWT, RBAC, Firebase**

I. Introduction

Modern digital collaboration requires security at multiple layers: encrypted storage for files at rest, secure communication channels for messaging, and access-controlled workspaces for team collaboration. While individual solutions exist, integrated educational platforms demonstrating security from system calls to web APIs remain rare.

This work presents a comprehensive platform addressing this gap through three subsystems: (1) a native C encryption tool demonstrating POSIX primitives, (2) a real-time encrypted chat application, and (3) a full-stack secure vault with enterprise security patterns. Each component demonstrates different aspects of operating system security while maintaining conceptual unity.

A. Motivation

Security education benefits from seeing how low-level primitives connect to high-level abstractions. Practical systems require coordinated protection at multiple layers. Understanding both native code and web security provides complete perspective on modern secure system design.

B. Contributions

1) Native encryption tool with dual interfaces (TUI/CLI) demonstrating POSIX system calls and binary-safe operations
2) Encrypted messaging platform with real-time synchronization and channel management using Firebase
3) Secure vault implementing JWT authentication, RBAC, audit trails, secure sharing, and lifecycle management
4) Unified demonstration of OS concepts: system calls, file descriptors, process lifecycle, concurrency control, secure storage
5) Comprehensive testing validating encryption correctness across all subsystems

II. Related Work

Traditional encryption tools like GPG [10] and OpenSSL [11] provide robust cryptography but complex interfaces. Educational tools often lack system-level integration [6]. Signal Protocol [12] established modern encrypted messaging standards, while Firebase-based solutions [8] simplify real-time synchronization but typically lack built-in encryption. Enterprise platforms like Dropbox provide secure storage but rely on server-side encryption and centralized trust models.

This work synthesizes these domains into a cohesive platform demonstrating OS-level security primitives alongside application-layer patterns, serving both educational and practical purposes.

III. System Architecture

A. Overview

The platform consists of three independent but conceptually unified subsystems (Table I).

TABLE I
System Architecture Overview

Component	Technology Stack
Native Tool	C with POSIX + ncurses; Caesar/XOR ciphers
CipherChat	React + Firebase Realtime Database; Web Crypto API encryption
CipherVault	React + Flask + PostgreSQL; AES-256-GCM, JWT, RBAC

B. Design Principles

Defense in Depth: Security at multiple layers—encryption at rest and in transit, authentication/authorization, audit logging.

Separation of Concerns: Each subsystem is independently deployable and testable.

Principle of Least Privilege: Granular RBAC with distinct permission levels.

OS Integration: Native tool demonstrates kernel interaction; web components show user-space patterns.

IV. Native Encryption Tool

A. Architecture and Components

The native tool is implemented in C with modular architecture separating encryption logic, file I/O operations, and user interfaces (Table II).

TABLE II
Native Tool Component Structure

Module	Responsibility
`main.c`	Entry point, argument parsing, mode dispatch
`encryption.c`	Caesar and XOR cipher implementation
`file_io.c`	POSIX file operations, buffer management
`ui.c`	Ncurses TUI rendering and interaction

B. File I/O Implementation

The implementation uses direct POSIX system calls (`open`, `read`, `write`, `close`) rather than standard C library functions, demonstrating the user-kernel boundary. File descriptors are managed explicitly with comprehensive error checking at each operation. A fixed-size buffer (4KB) enables constant-memory streaming of arbitrarily large files, demonstrating efficient resource management.

C. Cryptographic Algorithms

1) Caesar Cipher: Byte-level substitution cipher with modular arithmetic:

$$E(b,k) = (b+k) \bmod 256, \quad D(c,k) = (c-k) \bmod 256 \tag{1}$$

where b is plaintext byte, c is ciphertext byte, and $k \in [0,255]$ is the integer key. The modulo 256 operation ensures proper wrapping for byte values.

2) XOR Cipher: Self-inverse bitwise exclusive-OR with repeating key stream:

$$E(b_i,K) = D(c_i,K) = b_i \oplus K_{i \bmod |K|} \tag{2}$$

where K is the variable-length key string. This property simplifies implementation while demonstrating symmetric encryption fundamentals.

Both algorithms support binary-safe processing, handling any file type without corruption.

D. Dual Interface Design

Interactive TUI: The ncurses-based terminal user interface provides color-coded visual feedback (green for success, red for errors, cyan for information), ASCII progress bars, and arrow key navigation. Initialization establishes proper terminal modes including raw input, no echo, and keypad support.

Scriptable CLI: A flag-based command-line interface using `getopt` supports batch processing and shell integration. The interface enables pipeline composition, automated workflows, and conditional execution. Example usage: `./encrypt_tool -e -a xor -k "secret" -i file.txt -o file.enc`.

E. Operating System Concepts

System Call Overhead: Each file operation triggers context switching from user mode to kernel mode, demonstrating privilege separation inherent in modern OS design.

File Descriptor Lifecycle: Explicit management covers allocation, usage, and cleanup, including proper error path handling to prevent resource leaks.

Process Lifecycle: Clear phases map to fork-exec-wait models with initialization, execution, and termination.

I/O Redirection: CLI design supports Unix pipeline composition and stream redirection through stdin/stdout handling.

V. CipherChat: Encrypted Messaging

A. Architecture

CipherChat implements a React-based single-page application with Firebase Realtime Database backend. The architecture separates presentation logic, state management, and data synchronization layers.

B. Security Model

1) End-to-End Encryption: Messages are encrypted on the client using the Web Crypto API before transmission to Firebase. The encryption workflow: (1) client generates or retrieves channel-specific encryption key, (2) message encrypted using AES-GCM with random IV, (3) ciphertext, IV, and authentication tag stored in Firebase, (4) recipients decrypt locally using shared key. This ensures the Firebase server never accesses plaintext messages.

2) Channel Management: Each communication channel possesses a unique encryption key. Users must possess the channel key to decrypt messages. The system supports both public channels with shared keys and private channels requiring secure key exchange. Channel metadata (names, member lists, creation timestamps) is stored separately from encrypted message content.

3) Presence Tracking: Real-time user status leverages Firebase's connection state monitoring. The system automatically detects disconnections and updates user status, providing online/offline indicators with minimal latency. Presence information uses Firebase's `onDisconnect` callback mechanism for reliable cleanup.

C. Features

The platform supports real-time message synchronization with sub-second delivery latency, channel-based organization for multiple conversation spaces, Firebase Authentication integration for user management, persistent encrypted message history, and rich text rendering including Markdown formatting and emoji support.

VI. CipherVault: Secure Collaboration

A. Architecture

CipherVault implements a full-stack architecture with React frontend, Flask REST API backend, and PostgreSQL database. The separation enables independent scaling of presentation and business logic layers.

B. Cryptographic Implementation

1) Key Derivation: User passwords are strengthened using PBKDF2-HMAC-SHA256 with 480,000 iterations, meeting OWASP 2023 recommendations. Each user receives a unique 32-byte random salt. The mathematical formulation:

$$K = \text{PBKDF2}(P, S, 480000, 32) \tag{3}$$

where P is user passphrase, S is random salt, and K is derived 256-bit key.

2) Authenticated Encryption: Files are encrypted using AES-256-GCM, providing both confidentiality and integrity in a single operation:

$$(C, T) = \text{AES-GCM}(K, IV, M, AD) \tag{4}$$

where K is 256-bit key, IV is 96-bit random nonce, M is plaintext file content, AD is authenticated metadata, C is ciphertext, and T is 128-bit authentication tag. GCM mode detects any tampering through tag verification during decryption.

3) Integrity Verification: Additional SHA-256 hashing enables integrity verification without decryption:

$$H = \text{SHA256}(M) \tag{5}$$

The hash is stored separately, allowing quick integrity checks before expensive decryption operations.

C. Authentication System

1) JWT-based Authentication: The system implements stateless authentication using JSON Web Tokens. Access tokens have 15-minute lifetime while refresh tokens last 7 days. Token rotation occurs on refresh requests, invalidating old tokens. Tokens are stored in secure HttpOnly cookies, preventing JavaScript access and XSS attacks. Each token includes user ID, assigned roles, and expiration timestamp. The backend validates both signature and expiration on every request.

D. Role-Based Access Control

The system implements a four-tier role hierarchy (Table III). Access control is enforced at the API layer through decorator-based permission checks, providing reusable authorization patterns across endpoints.

TABLE III
CipherVault Role Hierarchy

Role	Permissions
Owner	Full control: manage members, delete room, configure settings
Admin	Add/remove members, upload/delete files, edit metadata
Member	Upload files, download files, add comments
Viewer	Read-only: view files and metadata

E. Advanced Features

1) Audit Logging: The system maintains comprehensive activity logs tracking authentication events (login, logout, failed attempts), file operations (upload, download, delete, share), permission changes (role assignments, member additions), and administrative actions (room creation, settings changes). Each log entry includes timestamp, user ID, IP address, action type, outcome, and contextual details. Logs use append-only storage to prevent tampering.

2) Secure File Sharing: Two sharing mechanisms are supported: (1) expiring links with time-limited access tokens, optional password protection, automatic cleanup after expiration, and usage tracking; (2) room-based sharing with collaborative workspaces, inherited permissions from room roles, collaborative metadata editing, and shared audit trails.

3) Concurrency Control: File operations use PostgreSQL advisory locks to prevent race conditions. The implementation provides read locks for concurrent reads and write locks for exclusive access. Automatic lock timeouts (5 minutes) and lock queues ensure fairness. Advisory locks are preferred over table-level locks for fine-grained control.

4) Lifecycle Management: A background scheduler handles expired file deletion, secure deletion with multi-pass overwrite following DoD 5220.22-M standard (7-pass), orphaned file detection and cleanup, and storage quota enforcement per user and room. The secure deletion process overwrites file content with zeros, ones, and five passes of random data before unlinking, preventing data recovery.

VII. Implementation

A. Codebase Statistics

The implementation spans 10,292 lines of code across all subsystems (Table IV).

TABLE IV
IMPLEMENTATION SCALE (LINES OF CODE)

Component	LOC
Native Tool (C)	1,149
CipherChat Frontend	2,430
CipherChat Backend	245
CipherVault Frontend	3,432
CipherVault Backend	3,036
Total	**10,292**

B. Technology Stack

TABLE V
COMPLETE TECHNOLOGY STACK

Layer	Technologies
Native Tool	C (C99), GNU Make, POSIX, ncurses
CipherChat Frontend	React 18, Material-UI, Web Crypto API
CipherChat Backend	Firebase Realtime DB, Firebase Auth
CipherVault Frontend	React 18, React Router v6, Axios
CipherVault Backend	Flask 2.x, Flask-JWT-Extended
Database	PostgreSQL 14, SQLAlchemy ORM
Cryptography	Python Cryptography 41.x

C. Build and Deployment

The native tool compiles with GNU Make on Linux, macOS, and Unix systems. CipherChat deploys as a static site (Netlify/Vercel) with Firebase hosting for backend services. CipherVault uses separate deployments for frontend (static site) and backend (Gunicorn/uWSGI with PostgreSQL instance).

VIII. Testing and Validation

A. Native Tool Validation

Automated roundtrip testing validates encryption correctness through encrypt-decrypt cycles with byte-level comparison. All test cases achieved 100% bit-accurate reconstruction (Table VI).

B. CipherChat Validation

End-to-end encryption was verified through message interception tests confirming server-side ciphertext storage. Real-time synchronization maintains median latency of 180ms. The system supports 50+ concurrent users per channel with message throughput exceeding 100 messages per second.

TABLE VI
NATIVE TOOL ROUNDTRIP TEST RESULTS

Test Case	Algorithm	Size	Status
Text file	Caesar	53 B	PASS
Text file	XOR	53 B	PASS
Binary (image)	Caesar	154 KB	PASS
Binary (image)	XOR	154 KB	PASS
Executable	Caesar	142 KB	PASS
Executable	XOR	142 KB	PASS

C. CipherVault Security Testing

Authentication: Token expiration is properly enforced. Refresh token rotation works correctly. Password hashing demonstrates timing-attack resistance.

Authorization: RBAC is enforced at the API boundary. Privilege escalation attempts are prevented. Cross-room access is properly isolated.

Encryption: AES-GCM correctly fails on tampered ciphertext through authentication tag verification. IV uniqueness is verified across all encryptions. PBKDF2 iteration count meets OWASP standards.

D. Performance Metrics

System performance characteristics are summarized in Table VII.

TABLE VII
SYSTEM PERFORMANCE CHARACTERISTICS

Metric	Value
Native Caesar throughput	200 MB/s
Native XOR throughput	180 MB/s
CipherChat message latency	180 ms (median)
CipherVault upload (10 MB)	2.3 s
CipherVault encryption overhead	15–20%

IX. Discussion

A. Educational Value

The platform demonstrates security implementation across three abstraction levels: (1) low-level system calls and file descriptors in the native tool, (2) mid-level real-time protocols and client-side encryption in CipherChat, (3) high-level web APIs, JWT, RBAC, and lifecycle management in CipherVault. This progression illustrates how security primitives at each layer build upon lower-level abstractions, providing students complete perspective on secure system design.

B. Operating System Concepts

The implementation demonstrates several fundamental OS concepts. System call overhead and context switching are evident in the native tool's POSIX operations. File descriptor lifecycle management includes proper error handling and resource cleanup. Process lifecycle follows clear initialization-execution-termination phases. Concurrency control through advisory locks prevents race conditions in multi-user scenarios. Secure storage with lifecycle policies manages resource allocation and reclamation. Session management through JWT tokens mirrors OS session concepts with timeout and renewal mechanisms.

C. Practical Applications

Beyond educational purposes, each subsystem serves practical functions. The native tool provides lightweight encryption for scripting and automation tasks. CipherChat enables secure team communication with minimal infrastructure requirements. CipherVault offers document management capabilities suitable for small teams and project collaboration.

D. Limitations

Native Tool: Caesar and XOR ciphers provide educational value but lack cryptographic security for production use. Key management via command-line arguments exposes keys in process listings. No support for asymmetric encryption limits use cases.

CipherChat: Key exchange mechanism requires out-of-band communication for initial setup. Lack of forward secrecy means compromised keys expose all historical messages. Firebase dependency limits deployment flexibility and server choice.

CipherVault: Server-side architecture requires trust in the server operator. The system does not implement true zero-knowledge architecture where the server cannot access user data. Single-server deployment limits horizontal scalability.

E. Future Work

Planned enhancements include integrating Signal Protocol for forward secrecy in messaging, adding RSA/ECC asymmetric encryption to the native tool, implementing Shamir secret sharing for distributed key recovery, supporting end-to-end encrypted file sharing in CipherVault, adding multi-device synchronization for seamless user experience, integrating hardware security modules for key protection, implementing blockchain-based audit log immutability, adding multi-threaded encryption for improved performance, and supporting WebAssembly for browser-based crypto acceleration.

X. Conclusion

This paper presented an integrated secure communication platform spanning three subsystems: a native C encryption tool demonstrating POSIX system calls and dual-mode interfaces, CipherChat providing real-time encrypted messaging with Firebase, and CipherVault implementing full-stack security with AES-256-GCM, JWT, RBAC, audit logging, and lifecycle management.

The platform successfully demonstrates security implementation from low-level system calls to high-level web APIs, providing both educational value and practical utility. Comprehensive testing validates encryption correctness with 100% roundtrip accuracy, proper security enforcement across authentication and authorization layers, and acceptable performance characteristics with native tool throughput of 200 MB/s and web platform encryption overhead of 15-20%.

The modular architecture with clear separation of concerns enables independent development and testing of each subsystem while maintaining conceptual coherence as a unified security platform. The project serves as a practical reference implementation for students learning operating system security concepts and developers prototyping secure communication workflows.

Future enhancements will focus on cryptographically robust algorithms, zero-knowledge architectures, multi-device synchronization, and performance optimization. The fundamental design principles—defense in depth, separation of concerns, and OS integration—provide a solid foundation for these extensions.

Acknowledgment

The authors acknowledge the faculty and peers at Rishihood University, Department of Computer Science and Artificial Intelligence, for their guidance and support. Special thanks to the open-source communities behind POSIX, ncurses, React, Firebase, Flask, and Python Cryptography library.

References

[1] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," FIPS Publication 197, 2001.
[2] D. McGrew and J. Viega, "The Galois/Counter Mode of Operation (GCM)," NIST Special Publication 800-38D, 2007.
[3] National Institute of Standards and Technology, "Recommendation for Password-Based Key Derivation," NIST Special Publication 800-132, 2010.
[4] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, 2015.
[5] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," FIPS Publication 180-4, 2015.
[6] OWASP Foundation, "Password Storage Cheat Sheet," 2024. [Online]. Available: https://cheatsheetseries.owasp.org/
[7] M. Kerrisk, *The Linux Programming Interface*. San Francisco: No Starch Press, 2010.
[8] Google, "Firebase Realtime Database Documentation," 2024. [Online]. Available: https://firebase.google.com/docs/database
[9] Pallets Project, "Flask Documentation," 2024. [Online]. Available: https://flask.palletsprojects.com/
[10] The GnuPG Project, "The GNU Privacy Guard," 2024. [Online]. Available: https://gnupg.org/
[11] OpenSSL Software Foundation, "OpenSSL: Cryptography and SSL/TLS Toolkit," 2024. [Online]. Available: https://www.openssl.org/
[12] M. Marlinspike and T. Perrin, "The X3DH Key Agreement Protocol," Open Whisper Systems, 2016.