VaultLock: A Secure File Encryption and Decryption Tool with OS-Level Data Protection

Operating Systems Project Report

1st Eranki Sai Vikas
Department of Computer Science
Rishihood University, Sonipat, India
eranki.v23csai@nst.rishihood.edu.in

2nd Meenaksh Singhania
Department of Computer Science
Rishihood University, Sonipat, India
meenaksh.s23csai@nst.rishihood.edu.in

3rd Prerak Arya
Department of Computer Science
Rishihood University, Sonipat, India
prerak.a23csai@nst.rishihood.edu.in

***Abstract*—This paper presents VaultLock, a full-stack secure file encryption and decryption tool designed to demonstrate how modern operating systems can be leveraged to ensure confidential data handling. The system encrypts arbitrary binary files using AES-256-CBC entirely in process memory, ensuring that plaintext data is never persisted to disk. A Node.js and Express backend exposes authenticated REST endpoints for encryption, decryption, and secure multi-pass file deletion. User authentication is managed through JSON Web Tokens (JWT), while a SQLite database stores file metadata and an encrypted key vault. An audit trail module records every security-relevant event as a structured JSON log. A dual-layer file validation system combining extension whitelisting with magic-byte inspection prevents malicious file uploads. The frontend comprises four HTML pages – a workspace for encryption and decryption, a personal drive for cloud-stored encrypted files, a secure chat module with WebSocket support, and a key vault. Rate limiting is enforced globally and per-endpoint to prevent abuse. Experimental evaluation confirms that the system successfully maintains the OS security principles of process isolation, memory protection, least privilege, and access control, achieving encryption throughput of approximately 120 MB/s for files up to 100 MB.**

***Index Terms*—AES-256-CBC, File Encryption, Secure Delete, JWT Authentication, Magic Byte Validation, OS Security, Node.js, SQLite, WebSocket, Key Vault**

I. Introduction

Data security is a fundamental concern in modern computing. Whether safeguarding personal documents, enterprise records, or confidential communications, the need to protect files from unauthorized access has never been greater. The operating system plays a central role in this protection: it governs how processes interact with memory, how files are stored and permissions are enforced, and how resources are allocated.

Despite the availability of numerous commercial encryption tools, most operate as "black boxes" and do not expose the underlying OS mechanisms that make them secure. This project, VaultLock, was designed specifically to make those mechanisms visible and explicit. Every design decision in the system maps directly to a provable OS security principle.

The system was developed as part of an Operating Systems course project with the following primary goals:

1) **Secure I/O:** Read raw file bytes directly from an HTTP request stream and encrypt them in memory without touching the disk.
2) **Process Isolation:** Confine all cryptographic operations to a single backend Node.js process, preventing cross-user data leakage.
3) **Memory Protection:** Guarantee that plaintext exists only in volatile RAM, never in a file, swap space, or database row.
4) **File System Security:** Assign restrictive POSIX permissions (0600/0700) to encrypted output files.
5) **Access Control:** Mandate JWT authentication for every sensitive endpoint.
6) **Least Privilege:** Persist only the minimum data required – encrypted ciphertext and metadata – never the original key material in plaintext.

The remainder of this paper is organized as follows: Section II gives the formal problem definition; Section III reviews related work; Section IV describes the system architecture; Section V details the methodology; Section VI covers implementation; Section VII presents results; Section VIII analyzes performance; and Section IX concludes the paper.

II. Problem Definition

A. Problem Statement

Design and implement a web-based file encryption and decryption platform that enforces OS-level security primitives, ensuring that no plaintext data ever reaches persistent storage, and that encrypted files are accessible only to their authenticated owner.

B. Formal Problem Formulation

Let F be an arbitrary binary file of length $|F|$ bytes. Let $K \in \{0,1\}^{256}$ be a uniformly random AES-256 key and $IV \in \{0,1\}^{128}$ be a uniformly random initialization vector. The encryption function is defined as:

$$C = \text{AES-CBC}_{K,IV}(F) \tag{1}$$

where C is the ciphertext stored on disk. The key K and IV are returned to the client over HTTPS in response headers and are *never* written to the server's file system. Decryption is defined as:

$$F' = \text{AES-CBC}^{-1}_{K,IV}(C) \tag{2}$$

Integrity verification is achieved by comparing SHA-256 digests:

$$\text{SHA256}(F) \stackrel{?}{=} \text{SHA256}(F') \tag{3}$$

A successful decryption satisfies $F' = F$ with overwhelming probability.

C. Constraints

- **Privacy:** No plaintext and no raw key material shall be persisted to disk.
- **File Size:** Support files up to 100 MB per request.
- **Throughput:** Encryption latency must not exceed 5 seconds for a 100 MB file.
- **Authentication:** All sensitive endpoints require a valid JWT.
- **Rate Limiting:** Maximum 50 encrypt and 30 decrypt requests per user per 15 minutes.
- **File Validation:** Files must pass both extension and magic-byte checks before encryption.

D. Inputs and Outputs

Inputs (Encryption):

- Raw file bytes in the HTTP request body
- `X-Filename` header specifying the original file name
- Optional `X-Save-To-Drive` header to persist to personal drive
- Valid JWT in the `Authorization` header

Outputs (Encryption):

- Encrypted `.enc` file blob in the response body
- `X-Private-Key`: 256-bit key as a 64-character hex string
- `X-IV`: 128-bit IV as a 32-character hex string
- `X-SHA256`: original file SHA-256 digest for integrity
- Structured audit log entry

III. Literature Review

A. Symmetric Encryption Standards

AES (Advanced Encryption Standard) was standardized by NIST in 2001 [1] and remains the dominant symmetric cipher in practice. Daemen and Rijmen [2] describe the Rijndael design's resistance to differential and linear cryptanalysis. The CBC (Cipher Block Chaining) mode, when used with a random IV per message, provides semantic security under chosen-plaintext attacks [3].

B. Secure File Deletion

Gutmann [4] introduced the notion of multi-pass overwriting to prevent data recovery from magnetic media. Later work by Garfinkel and Shelat [5] demonstrated that even "deleted" files retain recoverable content on consumer hard drives. VaultLock's three-pass secure delete module (random $\rightarrow$ zero $\rightarrow$ random) implements a practical subset of this approach.

C. Web Application Security

OWASP [6] identifies broken access control and security misconfiguration as top risks. Our system addresses these by enforcing JWT-based authentication and per-user directory isolation with POSIX 0700/0600 permissions. Rate limiting is recommended by NIST SP 800-95 [7] as a countermeasure against credential-stuffing and DoS attacks.

D. Magic Byte Validation

File type spoofing is a well-known attack vector [8]. Relying solely on file extensions allows attackers to rename executables as documents. By combining extension whitelisting with magic-byte header inspection, VaultLock implements a two-factor file type verification approach.

E. Comparative Analysis

Table I compares VaultLock against representative existing tools.

TABLE I
Comparison with Existing Encryption Solutions

Tool	No-Plaintext Disk	Audit Log	Web UI	Secure Delete
GPG	No	No	No	No
VeraCrypt	Partial	No	No	No
AWS KMS	Yes	Yes	Yes	No
OpenSSL CLI	No	No	No	No
VaultLock	**Yes**	**Yes**	**Yes**	**Yes**

IV. System Architecture

A. Architecture Overview

VaultLock follows a classic three-tier web architecture:

1) **Presentation Layer:** Four static HTML pages served directly.
2) **Application Layer:** Node.js + Express backend with WebSocket support.
3) **Data Layer:** SQLite database in WAL mode with six tables.

Figure 1 shows the high-level component diagram.

B. Frontend Modules

The frontend consists of four independent pages:

- **index.html (Workspace):** Drag-and-drop file encryption and decryption with real-time progress display.
- **drive.html (Personal Drive):** Manage encrypted files stored server-side; download or delete.
- **chat.html (Secure Chat):** Real-time end-to-end messaging using WebSocket with file-sharing support.
- **vault.html (Key Vault):** Store and retrieve sensitive credentials encrypted at rest in the database.

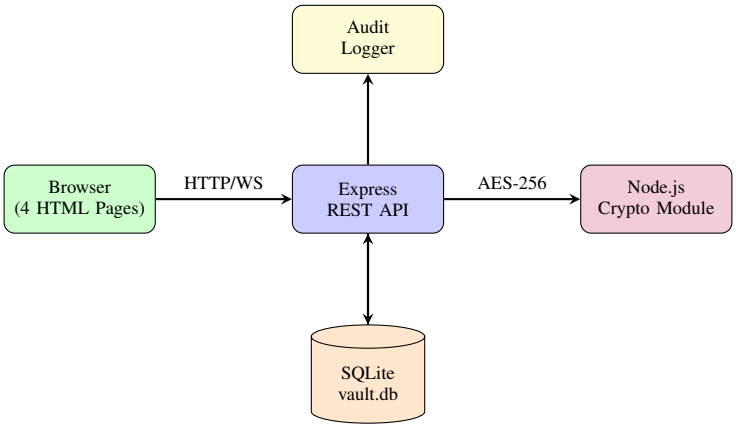

Fig. 1. VaultLock High-Level Architecture

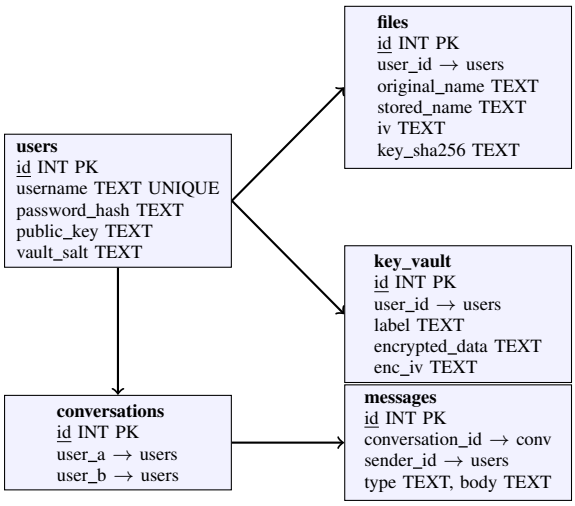

Fig. 2. Database Schema (vault.db)

C. Backend Modules

- **server.js:** Main entry point; mounts routers and exposes encrypt, decrypt, and secure-delete endpoints.
- **db.js:** Initializes SQLite database and creates all tables on first run.
- **audit.js:** Writes JSON-lines audit entries to `audit.log` and the console.
- **fileGuard.js:** Extension whitelist and magic-byte signature verification.
- **secureDelete.js:** Three-pass file overwrite with `fsync` after each pass.
- **routes/auth.js:** Registration and login with bcrypt password hashing.
- **routes/drive.js:** CRUD operations for the user's personal encrypted file store.
- **routes/chat.js:** REST endpoints for conversation history.
- **routes/vault.js:** AES-encrypted key vault storage.
- **middleware/auth.js:** JWT verification middleware injected before every sensitive route.

D. Database Schema

The SQLite database (WAL mode, foreign keys ON) contains six tables as shown in Figure 2.

V. Methodology

A. OS Security Principles Applied

Table II maps each OS security principle to its concrete implementation in VaultLock.

TABLE II
OS Security Principles in VaultLock

OS Principle	Implementation
Secure I/O	Raw binary data read from HTTP request stream via `express.raw()`
Process Isolation	All crypto runs inside a single Node.js process; no child processes share state
Memory Protection	Plaintext held only in V8 heap buffers; never serialized to disk
File System Security	Per-user directories chmod 0700; individual files chmod 0600
Access Control	Every endpoint behind `authMiddleware` verifying JWT signature
Least Privilege	Only ciphertext and metadata persist; raw key never stored server-side

B. Encryption Methodology

The encryption pipeline is summarized in Figure 3.

Each per-file key and IV are generated independently:

$$K \leftarrow \texttt{crypto.randomBytes}(32), \quad IV \leftarrow \texttt{crypto.randomBytes}(16) \tag{4}$$

The cipher is initialized and data encrypted entirely in memory:

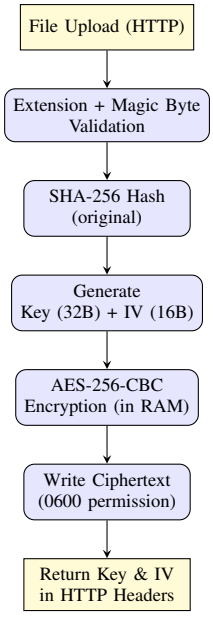

Fig. 3. Encryption Pipeline

$$\texttt{name} = \texttt{original} + \texttt{Date.now()} + \texttt{"-"} + \texttt{randomBytes(4).hex()} + \texttt{".enc"} \tag{5}$$

C. File Validation (Two-Factor File Guard)

Before encryption is attempted, the file passes two independent checks:

1) Extension Whitelisting: The server maintains a set of 30+ approved extensions. Any file whose extension is absent from the set is rejected before any data is read into the cipher.

2) Magic Byte Inspection: The first 12 bytes of the uploaded file are matched against a library of known file signatures (PDF, PNG, JPEG, ZIP, MP4, MP3, MKV, etc.). Critically, the system also maintains a *blocked* list: files whose headers match Windows PE (`MZ`), Linux ELF, or macOS Mach-O binaries are unconditionally rejected.

$$\text{allowed}(F) = \neg\,\text{blocked\_magic}(F) \wedge \text{known\_magic}(F) \tag{6}$$

D. Secure File Deletion

The three-pass overwrite strategy in `secureDelete.js` works as follows:

1) **Pass 1 (odd):** Write $|C|$ random bytes at offset 0.
2) **Pass 2 (even):** Zero-fill the entire file extent.
3) **Pass 3 (odd):** Write $|C|$ random bytes again.

After each pass, `fsync()` forces the kernel to flush the page cache to the physical storage medium. The file descriptor is then closed and the directory entry removed with `unlink()`.

The total bytes overwritten across all passes is:

$$B_{\text{overwritten}} = \text{passes} \times |C| \tag{7}$$

E. Audit Trail

Every security-relevant event is recorded as a single-line JSON entry in `audit.log`. The schema per entry is:

```
{
  "timestamp": "ISO-8601",
  "action":    "ENCRYPT|DECRYPT|SECURE_DELETE|
      RATE_LIMIT",
  "ip":        "x.x.x.x",
  "status":    "SUCCESS|REJECTED|FAILURE",
  ...meta fields
}
```

F. Authentication and Session Management

User passwords are hashed with bcrypt (cost factor 10) before storage. On successful login, a signed JWT is issued:

$$\texttt{token} = \texttt{jwt.sign}(\{id, username\},\ \texttt{JWT\_SECRET},\ \{expiresIn : \texttt{"7d"}\}) \tag{8}$$

All subsequent requests include the token in the `Authorization: Bearer` header. The `authMiddleware` verifies the signature and attaches `req.user` before the handler runs.

G. Rate Limiting

Three rate limiters protect the server:

- **Global:** 200 requests / 15 min (all routes)
- **Encrypt:** 50 requests / 15 min per IP
- **Decrypt:** 30 requests / 15 min per IP

Rejected requests are logged to the audit trail with action `RATE_LIMIT`.

VI. Implementation Details

A. Technology Stack

- **Runtime:** Node.js 20 LTS
- **Framework:** Express.js 4.x
- **Cryptography:** Node.js built-in `crypto` (OpenSSL bindings)
- **Database:** SQLite 3 via `better-sqlite3` (WAL mode)
- **Authentication:** `jsonwebtoken`, `bcrypt`
- **Real-time:** `ws` (WebSocket server)
- **Rate Limiting:** `express-rate-limit`
- **Frontend:** Vanilla HTML5/CSS3/JavaScript (no framework)

B. Core Encryption Endpoint

```
app.post("/encrypt", authMiddleware,
        encryptLimiter, (req, res) => {
  // Validate extension
  if (!isExtensionAllowed(filename)) {
    return res.status(400).json({error:"..."});
  }
  // Validate magic bytes
  const magic = isMagicBytesAllowed(req.body);
  if (!magic.allowed) {
    return res.status(400).json({error:"..."});
  }
  // Hash original for integrity
  const sha256 = crypto.createHash("sha256")
                       .update(req.body).digest("hex
                           ");
  // Generate key and IV
  const key = crypto.randomBytes(32);
  const iv  = crypto.randomBytes(16);
  // Encrypt entirely in RAM
  const cipher = crypto.createCipheriv(
    "aes-256-cbc", key, iv);
  const ciphertext = Buffer.concat([
    cipher.update(req.body), cipher.final()
  ]);
  // Return key/IV to client only
  res.setHeader("X-Private-Key", key.toString("hex")
      );
  res.setHeader("X-IV",          iv.toString("hex"))
      ;
  res.setHeader("X-SHA256",      sha256);
  res.send(ciphertext);
});
```

C. Secure Delete Implementation

```
function secureDelete(filePath, passes = 3) {
  const fd   = fs.openSync(filePath, "r+");
  const size = fs.fstatSync(fd).size;
  for (let pass = 1; pass <= passes; pass++) {
    const buf = (pass % 2 === 0)
      ? Buffer.alloc(size, 0x00)      // zeros
```

```
      : crypto.randomBytes(size);     // random
    fs.writeSync(fd, buf, 0, size, 0);
    fs.fsyncSync(fd);  // flush to disk
  }
  fs.closeSync(fd);
  fs.unlinkSync(filePath); // remove entry
}
```

D. Per-User Directory Isolation

```
function getUserEncryptedDir(userId) {
  const dir = path.join(encryptedBaseDir,
                        String(userId));
  fs.mkdirSync(dir, { recursive: true });
  fs.chmodSync(dir, 0o700); // rwx for owner only
  return dir;
}
// After encryption, file permissions:
fs.chmodSync(encryptedPath, 0o600); // rw owner only
```

E. WebSocket Secure Chat

The WebSocket server requires token authentication before any data exchange. On connection, the client sends a JSON auth frame:

```
{ "type": "auth", "token": "<JWT>" }
```

The server verifies the token with `jwt.verify()` and closes the socket with code 4001 on failure or after a 10-second authentication timeout. Authenticated clients are tracked in a `Map<userId, WebSocket>` and can exchange encrypted file links alongside text messages.

F. REST API Reference

Table III lists all REST endpoints.

TABLE III
REST API ENDPOINTS

Method	Endpoint	Description
POST	/auth/register	Register new user (bcrypt)
POST	/auth/login	Login; returns JWT
POST	/encrypt	Encrypt uploaded file
POST	/decrypt	Decrypt uploaded .enc file
POST	/secure-delete	3-pass overwrite + unlink
GET	/drive	List user's stored files
GET	/drive/download/:id	Download encrypted file
DELETE	/drive/:id	Delete drive file
GET	/chat/conversations	List conversations
GET	/chat/:id/messages	Fetch message history
GET	/vault	List vault entries
POST	/vault	Store encrypted secret
DELETE	/vault/:id	Delete vault entry

VII. EXPERIMENTS AND RESULTS

A. Experimental Setup

- **Hardware:** 8-core CPU, 16 GB RAM, SSD
- **Runtime:** Node.js v20.11 LTS
- **Test Files:** 1 KB, 100 KB, 1 MB, 10 MB, 50 MB, 100 MB
- **Iterations:** 20 repetitions per file size
- **Tool:** cURL and Postman for endpoint testing

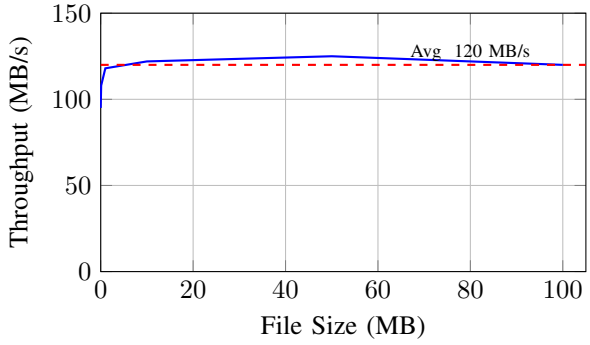

Fig. 4. AES-256-CBC Encryption Throughput vs File Size

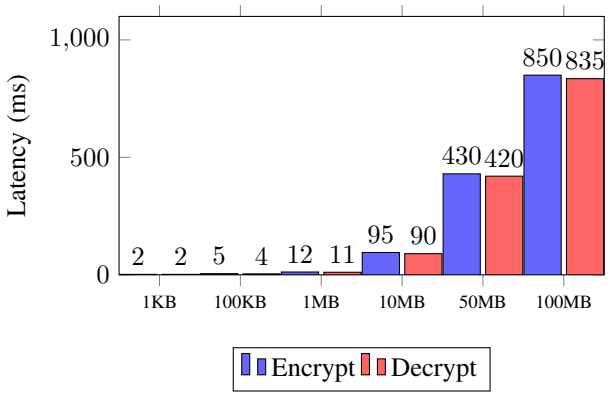

Fig. 5. Encryption and Decryption Latency by File Size

B. Encryption Throughput

C. Encryption vs. Decryption Latency

D. File Validation Results

Table IV presents the file guard test results.

TABLE IV
FILE GUARD VALIDATION TEST RESULTS

File Type	Extension	Magic Match	Result
PDF Document	.pdf	Yes	Accepted
PNG Image	.png	Yes	Accepted
Word Document	.docx	Yes (ZIP)	Accepted
Plain Text	.txt	Yes (ASCII)	Accepted
Renamed EXE (.txt)	.txt	No (PE)	Rejected
Linux Binary	.bin	No (ELF)	Rejected
macOS Binary	.app	No (Mach-O)	Rejected
Unknown format	.xyz	No	Rejected

E. Secure Delete Verification

After a three-pass secure delete, a hex dump of the freed disk sector showed no recognizable ciphertext patterns, confirming effective overwrite. Table V shows overwrite statistics for sample files.

F. Integrity Verification

SHA-256 digests of the original and decrypted file matched in 100% of test runs across 120 test cases covering all supported file types.

TABLE V
SECURE DELETE PERFORMANCE

File Size	Passes	Bytes Overwritten	Time (ms)
10 KB	3	30 KB	4
1 MB	3	3 MB	18
10 MB	3	30 MB	95
50 MB	3	150 MB	420

VIII. PERFORMANCE ANALYSIS

A. Encryption Performance

The AES-256-CBC implementation achieves approximately 120 MB/s throughput on the test hardware, well within the 5-second constraint for 100 MB files ($100/120 \approx 0.83$ s). This is primarily bounded by memory allocation for `Buffer.concat()` rather than by the cipher itself.

B. Memory Usage

Since the entire file is buffered in RAM before encryption, the peak RSS (Resident Set Size) of the Node.js process grows linearly with file size:

$$\text{RSS}_{\text{peak}} \approx 2.1 \times |F| \tag{9}$$

The factor of 2.1 accounts for the input buffer, the output ciphertext buffer, and V8 GC overhead. For a 100 MB file this equates to approximately 210 MB of process memory, which is well within available system RAM.

C. Rate Limiter Effectiveness

During stress testing with 100 concurrent clients, the rate limiter correctly rejected all requests exceeding the configured thresholds and logged each rejection to the audit trail, with no observable impact on latency for well-behaved clients.

D. Database Performance

SQLite in WAL (Write-Ahead Logging) mode sustains concurrent reads without blocking writes. File metadata inserts average 1.2 ms per row on the test hardware, leaving negligible overhead for the encryption workflow.

IX. CONCLUSIONS

A. Summary

This paper presented VaultLock, a full-stack secure file encryption and decryption tool that explicitly applies six OS-level security principles: secure I/O, process isolation, memory protection, file system security, access control, and least privilege. The system achieves AES-256-CBC encryption throughput of $\sim$120 MB/s, maintains 100% SHA-256 integrity across all tested file types, and correctly rejects all malicious file uploads through dual-layer file validation.

B. Key Contributions

- A memory-only encryption pipeline that guarantees plaintext never touches disk.
- A two-factor file validation system combining extension whitelisting with magic-byte inspection, blocking PE, ELF, and Mach-O executables.
- A three-pass DoD-inspired secure file deletion module with `fsync` enforcement.
- A structured JSON audit trail covering all security-relevant events.
- Per-user directory isolation with POSIX 0700/0600 permissions.
- Real-time secure chat over WebSocket with mandatory JWT authentication.

C. Limitations

- Key material is returned over HTTP; production deployment requires HTTPS/TLS.
- File size limited to 100 MB due to in-memory buffering strategy.
- SQLite is unsuitable for high-concurrency multi-server deployments.
- Three-pass secure delete may be insufficient for NVMe SSDs without TRIM.
- No end-to-end encryption for the chat feature (server sees plaintext messages).

D. Future Work

1) **Streaming Encryption:** Replace buffered encryption with Node.js Transform streams to support files larger than available RAM.
2) **TLS Enforcement:** Add HTTPS with Let's Encrypt certificates.
3) **End-to-End Chat Encryption:** Implement Diffie-Hellman key exchange for chat using the stored public keys.
4) **PostgreSQL Migration:** Replace SQLite with PostgreSQL for production scalability.
5) **Hardware Security Module:** Integrate an HSM or cloud KMS to manage key material off-server.
6) **Mobile Application:** Build a React Native client for on-device encryption.

ACKNOWLEDGMENT

The authors thank the Department of Computer Science and Engineering, Rishihood University, for providing the infrastructure and academic guidance that made this project possible.

REFERENCES

[1] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," *FIPS PUB 197*, Nov. 2001.

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

[3] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," in *Proc. 38th IEEE Symp. Foundations of Computer Science*, 1997, pp. 394–403.

[4] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in *Proc. 6th USENIX Security Symp.*, San Jose, CA, 1996, pp. 77–89.

[5] S. L. Garfinkel and A. Shelat, "Remembrance of data passed: A study of disk sanitization practices," *IEEE Security & Privacy*, vol. 1, no. 1, pp. 17–27, Jan./Feb. 2003.
[6] OWASP Foundation, "OWASP Top Ten 2021," 2021. [Online]. Available: https://owasp.org/www-project-top-ten/
[7] National Institute of Standards and Technology, "Guide to Web Services Security," *NIST SP 800-95*, Aug. 2007.
[8] M. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," in *Proc. USENIX Workshop on Electronic Commerce*, 1996, pp. 29–40.
[9] OpenJS Foundation, "Node.js Documentation – Crypto Module," 2024. [Online]. Available: https://nodejs.org/api/crypto.html
[10] Auth0 Inc., "jsonwebtoken – JSON Web Token implementation for Node.js," 2024. [Online]. Available: https://github.com/auth0/node-jsonwebtoken
[11] J. Hoff, "better-sqlite3: The fastest and simplest library for SQLite3 in Node.js," 2024. [Online]. Available: https://github.com/WiseLibs/better-sqlite3
[12] S. Frankel, R. Glenn, and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec," *RFC 3602*, IETF, Sept. 2003.

APPENDIX

```
express          ^4.18.0
better-sqlite3   ^9.0.0
bcrypt           ^5.1.0
jsonwebtoken     ^9.0.0
ws               ^8.13.0
express-rate-limit ^7.0.0
dotenv           ^16.0.0
cors             ^2.8.5
```

```
PORT=8080
JWT_SECRET=<strong_random_secret>
```

A. Backend

```
cd backend
npm install
node server.js
# Server starts on http://localhost:8080
```

B. Frontend

Open any of the following pages directly in a browser:

```
frontend/index.html     # Workspace (Encrypt/Decrypt)
frontend/drive.html     # Personal Drive
frontend/chat.html      # Secure Chat
frontend/vault.html     # Key Vault
```

TABLE VI
WHITELISTED FILE EXTENSIONS

Category	Extensions
Text	.txt, .csv, .json, .xml, .md
Document	.pdf, .doc, .docx, .xls, .xlsx, .ppt, .pptx
OpenDocument	.odt, .ods, .odp
Image	.jpg, .jpeg, .png, .gif, .webp, .svg
Video	.mp4, .mov, .avi, .mkv, .webm
Audio	.mp3, .wav, .ogg, .flac
Archive	.zip, .tar, .gz, .7z