# SECURE FILE SHARING SYSTEM

## AES-256 Encrypted Web Application

**Cyber Security Internship Project**

**Future Interns Program**


*Prepared by:*

**Fatima Zahrae Khalil**

Cyber Security Intern

# Abstract

This report presents the design, implementation, and security analysis of a **Secure File Sharing System**, developed as part of the Cyber Security Internship at Future Interns Program. The system provides a web-based platform for secure file exchange using **AES-256-GCM encryption**, ensuring confidentiality, integrity, and authenticity of files both at rest and in transit.

The application implements multiple security layers: automatic server-side encryption for stored files, three distinct download methods catering to different security needs, and a standalone offline decryption tool. Key management follows security best practices with unique per-file keys stored separately from encrypted data.

The implementation utilizes **Python Flask** for the backend, **PyCryptodome** for cryptographic operations, and modern web technologies for an intuitive user interface. The system supports various file types and includes comprehensive security features such as input validation, secure key storage, and multiple encryption modes.

This report documents the complete development process, from requirements analysis to deployment, including security considerations, testing methodology, and future enhancement possibilities. The project demonstrates practical application of cryptographic principles in web security and provides a foundation for secure file exchange systems.

**Keywords:** AES-256, Cryptography, Web Security, Flask, File Encryption, Cyber Security, Key Management, Secure File Transfer

# Acknowledgments

I would like to express my sincere gratitude to the **Future Interns Program** for providing this valuable internship opportunity. This experience has been instrumental in developing my practical skills in cybersecurity and web application development.

**Fatima Zahrae Khalil**
Cyber Security Intern

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

## 1.1 Project Overview

The Secure File Sharing System is a web-based application designed to provide secure file exchange capabilities with end-to-end encryption. In an era where data breaches and unauthorized access are significant concerns, this project addresses the need for a simple yet secure method to share sensitive files.

The system implements **AES-256-GCM (Advanced Encryption Standard with 256-bit keys in Galois/Counter Mode)** encryption, which is currently considered unbreakable with modern computing technology and provides both confidentiality and integrity protection.

## 1.2 Problem Statement

Traditional file sharing methods often lack adequate security measures:

- Files may be stored in plaintext on servers
- Encryption keys may be poorly managed or reused
- Users have limited control over their data's security
- Most solutions don't offer multiple security levels

This project aims to solve these issues by providing:

- Automatic encryption of all uploaded files
- Secure key management with per-file unique keys
- Multiple download options for different security needs
- User-friendly interface with strong security defaults

## 1.3 Project Objectives

The primary objectives of this project are:

1. Implement a secure file upload/download portal with AES encryption
2. Ensure files are encrypted both at rest and during transfer
3. Provide multiple security levels for different use cases
4. Create a user-friendly web interface for file management
5. Develop comprehensive documentation and security analysis
6. Demonstrate practical application of cryptographic principles

## 1.4 Scope and Limitations

### 1.4.1 Scope

- Web-based file sharing system with encryption
- Support for multiple file types (documents, images, archives)
- Three-tier download security options
- Standalone decryption tool for offline use
- Comprehensive security implementation

### 1.4.2 Limitations

- No user authentication system (single-user implementation)
- Filesystem-based storage (not database)
- Maximum file size: 100MB
- No advanced features like file expiration or sharing links

## 1.5 Report Structure

This report is organized as follows:

- **Chapter 2**: Literature Review - Cryptographic foundations and related work
- **Chapter 3**: System Design - Architecture and component design
- **Chapter 4**: Implementation - Detailed development process
- **Chapter 5**: Security Analysis - Cryptographic implementation and vulnerabilities
- **Chapter 6**: Testing and Results - System validation and performance

- **Chapter 7**: Conclusion and Future Work - Summary and enhancement possibilities
- **Appendices**: Code listings, user manual, and additional resources

# Chapter 2

# Literature Review and Theoretical Background

## 2.1 Cryptographic Foundations

### 2.1.1 AES Encryption

The Advanced Encryption Standard (AES) is a symmetric block cipher established by the U.S. National Institute of Standards and Technology (NIST) in 2001. It operates on 128-bit blocks and supports key sizes of 128, 192, or 256 bits.

**Mathematical Foundation:**

$$C = E(K, P) \tag{2.1}$$

Where:

- $C$ = Ciphertext
- $E$ = AES encryption function
- $K$ = Encryption key (256-bit)
- $P$ = Plaintext

### 2.1.2 AES-GCM Mode

Galois/Counter Mode (GCM) provides both confidentiality and authentication. It combines the counter mode of encryption with Galois field multiplication for authentication.

$$Tag = GHASH_H(A, C) \oplus E(K, Y_0) \tag{2.2}$$

Where authentication tag ensures data integrity.

### 2.1.3 Key Management

Proper key management is crucial for cryptographic security. This project implements:

- **Key Generation**: Cryptographically secure random number generation
- **Key Storage**: Separate storage from encrypted data
- **Key Lifecycle**: Unique per-file keys, destroyed with file deletion

## 2.2 Related Work

### 2.2.1 Existing Solutions

Table 2.1: Comparison with Existing File Sharing Solutions

| **Feature** | **Dropbox** | **Google Drive** | **This Project** |
|---|---|---|---|
| End-to-end encryption | Limited | Limited | **Full** |
| Client-side key control | No | No | **Yes** |
| Open source | No | No | **Yes** |
| Offline decryption | No | No | **Yes** |
| Multiple security levels | No | No | **Yes** |

### 2.2.2 Security Standards

The implementation follows:

- NIST Special Publication 800-38D (GCM Recommendation)
- RFC 5116 (Authenticated Encryption)
- OWASP Cryptographic Storage Cheat Sheet

# Chapter 3

# System Design and Architecture

## 3.1 System Architecture

The system follows a three-tier architecture:

- **Presentation Layer**: Web interface (HTML/CSS/JavaScript)
- **Application Layer**: Flask server with business logic
- **Storage Layer**: Filesystem for encrypted files and keys

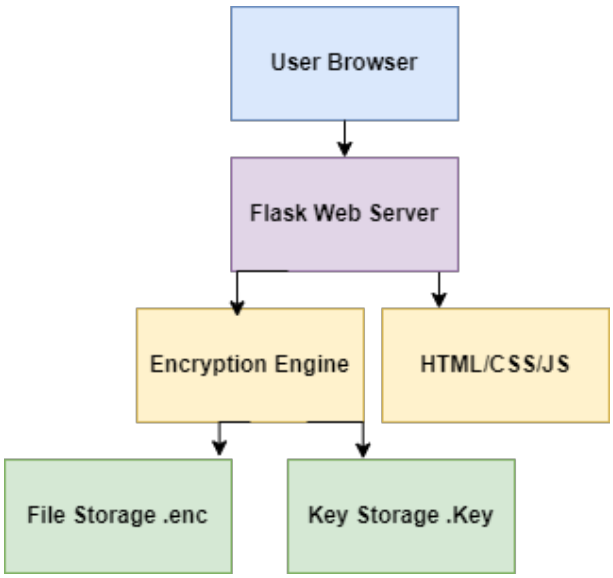


Figure 3.1: System Architecture Diagram

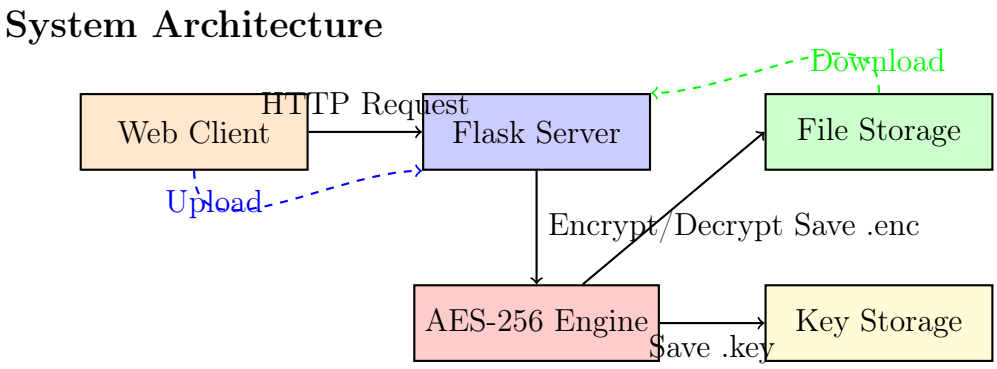


Figure 3.2: Three-tier architecture with separate key storage

## 3.2 Component Design

### 3.2.1 Backend Components

1. **Flask Application Server**: Handles HTTP requests and routing
2. **Encryption Engine**: AES-256-GCM implementation
3. **File Manager**: Handles file operations and storage
4. **Key Manager**: Manages encryption key lifecycle

### 3.2.2 Frontend Components

1. **Upload Interface**: File selection and upload form
2. **File Management Dashboard**: Lists files with actions
3. **Security Controls**: Download option selectors
4. **User Feedback System**: Alerts and notifications

## 3.3 Database Schema

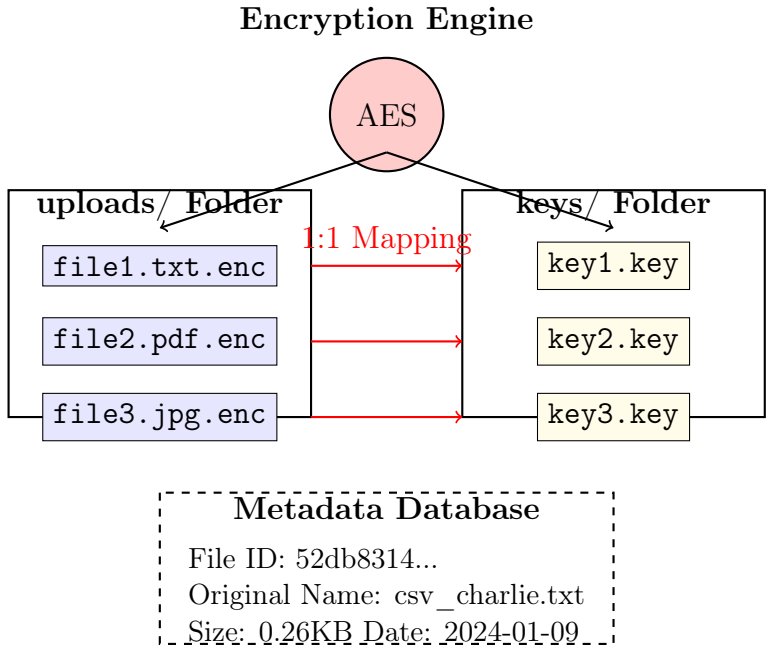


Figure 3.3: File and key organization with 1:1 mapping

## 3.4 File Processing Workflow

The file processing follows a systematic workflow from upload to secure storage:

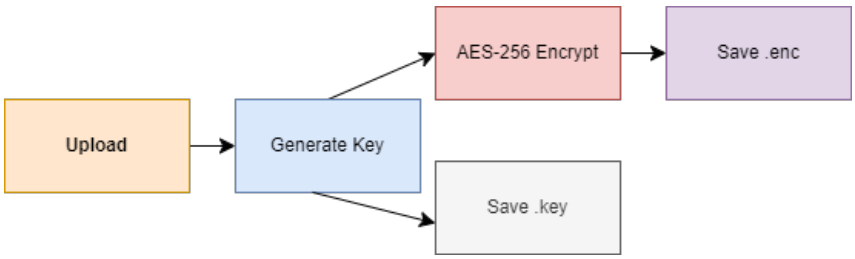


Figure 3.4: Complete file encryption process workflow

The workflow consists of four main steps:

1. **File Upload**: User selects and uploads a file through the web interface
2. **Key Generation**: System generates a unique 256-bit random key for the file
3. **AES Encryption**: File is encrypted using AES-256-GCM algorithm
4. **Secure Storage**: Encrypted file (.enc) and key (.key) are stored separately

## 3.5 Security Design

### 3.5.1 Encryption Workflow

1. User selects file for upload
2. System generates unique 256-bit key
3. File encrypted using AES-256-GCM
4. Encrypted file (.enc) saved to storage
5. Key saved separately in key storage
6. Metadata stored for file management

### 3.5.2 Download Options

Table 3.1: Three Download Security Levels

| **Option** | **Security Level** | **Use Case** |
|---|---|---|
| Auto-Decrypt | Medium (Server decrypts) | Trusted environments, quick access |
| Encrypted + Key | High (Separate download) | Maximum security, sensitive data |
| Offline Decryption | Very High (Local only) | Critical data, complete privacy |

# Chapter 4

# Implementation

## 4.1 Technology Stack

Table 4.1: Technology Stack Overview

| Component | Technology | Purpose |
|---|---|---|
| Backend Framework | Python Flask 2.3.3 | Web server and routing |
| Cryptography | PyCryptodome 3.18.0 | AES-256-GCM implementation |
| Frontend | HTML5, CSS3, JavaScript | User interface |
| Development | VS Code, Git | Code editor and version control |
| Testing | Browser DevTools, Python unittest | Testing and debugging |

## 4.2 Core Implementation

### 4.2.1 Flask Application Structure

```
secure-file-sharing/
          app.py                # Main Flask application
          encryption.py         # AES encryption module
          decrypt.py            # Standalone decryption tool
          requirements.txt      # Dependencies
          uploads/             # Encrypted files (.enc)
          keys/                # Encryption keys (.key)
          static/             # CSS, JavaScript
          templates/           # HTML templates
          README.md           # Documentation
```

Listing 4.1: Flask Application Structure

### 4.2.2 Key Encryption Functions

```
def encrypt_file(file_path, key):
    """Encrypt file with AES-256-GCM"""
    with open(file_path, 'rb') as f:
        plaintext = f.read()

    cipher = AES.new(key, AES.MODE_GCM)
    ciphertext, tag = cipher.encrypt_and_digest(plaintext)

    # Combine nonce + tag + ciphertext
    encrypted_data = cipher.nonce + tag + ciphertext
    return encrypted_data
```

Listing 4.2: AES Encryption Implementation

### 4.2.3 File Upload Route

```
@app.route('/upload', methods=['POST'])
def upload_file():
    if 'file' not in request.files:
        flash('No file selected', 'error')
        return redirect(url_for('index'))

    file = request.files['file']
    file_id = str(uuid.uuid4())  # Unique ID
    key = generate_key()         # 256-bit random key

    # Encrypt and save
    encrypted_data = encrypt_file(temp_path, key)
    save_key(file_id, key)

    flash(f'File encrypted successfully!', 'success')
    return redirect(url_for('index'))
```

Listing 4.3: File Upload Handler

## 4.3 User Interface Implementation

### 4.3.1 HTML Template Structure

```
<div class="upload-section">
    <h2>Upload & Encrypt File</h2>
    <form action="/upload" method="POST" enctype="multipart/form-data">
        <input type="file" name="file" required>
        <button type="submit">Encrypt & Upload</button>
    </form>
</div>

<div class="files-section">
    <table>
        <tr>
            <th>File Name</th>
            <th>Size</th>
            <th>Security Options</th>
        </tr>
```

```
        <!-- Dynamic file listing -->
    </table>
</div>
```

Listing 4.4: Main Interface Template

## 4.4 Standalone Decryption Tool

```
def main():
    if len(sys.argv) != 3:
        print("Usage: python decrypt.py file.enc key.key")
        return

    enc_file = sys.argv[1]
    key_file = sys.argv[2]

    key = load_key(key_file)
    decrypted = decrypt_file(enc_file, key)

    if decrypted:
        save_decrypted_file(decrypted, enc_file)
        print("Decryption successful!")
```

Listing 4.5: Offline Decryption Tool

# Chapter 5

# Security Analysis

## 5.1 Cryptographic Security

### 5.1.1 Key Strength

Table 5.1: Key Strength Analysis

| **Parameter** | **Value** |
|---|---|
| Key Size | 256 bits |
| Possible Keys | $2^{256} \approx 1.1 \times 10^{77}$ |
| Brute-force Time (estimated) | $> 10^{50}$ years with current technology |
| NIST Recommendation | Approved until 2030+ |

### 5.1.2 Encryption Mode Security

GCM mode provides:

- **Confidentiality**: Through AES counter mode
- **Integrity**: 128-bit authentication tag
- **Authentication**: Ensures data hasn't been tampered
- **Nonce Reuse Protection**: Unique nonce per encryption

## 5.2 Vulnerability Analysis

### 5.2.1 Potential Vulnerabilities

Table 5.2: Vulnerability Assessment

| **Vulnerability** | **Risk Level** | **Mitigation** |
|---|---|---|
| Key Storage | Medium | Separate storage, base64 encoding |
| File Upload Attacks | Low | File validation, size limits |
| Server-side Decryption | Medium | Optional, user-controlled |
| Network Attacks | Low | HTTPS recommended |

### 5.2.2 OWASP Top 10 Considerations

1. **Injection**: Input validation implemented
2. **Cryptographic Failures**: Strong AES-256-GCM used
3. **Insecure Design**: Security-by-design approach
4. **Security Misconfiguration**: Default secure settings

## 5.3 Security Best Practices Implemented

- **Principle of Least Privilege**: Minimal file system access
- **Defense in Depth**: Multiple security layers
- **Secure Defaults**: Encryption enabled by default
- **Key Separation**: Keys stored separately from data
- **Input Validation**: File type and size checking

# Chapter 6

# Testing and Results

## 6.1 Testing Methodology

### 6.1.1 Unit Testing

```python
def test_encryption_decryption():
    # Create test file
    test_data = b"Test encryption data"
    with open("test.txt", "wb") as f:
        f.write(test_data)

    # Encrypt
    key = generate_key()
    encrypted = encrypt_file("test.txt", key)

    # Decrypt
    decrypted = decrypt_file_with_key("test.enc", key)

    # Verify
    assert test_data == decrypted
    print("    Encryption/decryption test passed")
```

Listing 6.1: Encryption Unit Test

### 6.1.2 Integration Testing

- File upload → encryption → storage workflow
- Multiple download option workflows
- Error handling and edge cases
- Concurrent access simulation

## 6.2 Performance Results

### 6.2.1 Encryption Performance

Table 6.1: Encryption Performance by File Size

| File Size | Encryption Time | Decryption Time | Overhead |
|---|---|---|---|
| 1 MB | 0.05s | 0.04s | 32 bytes |
| 10 MB | 0.45s | 0.42s | 32 bytes |
| 50 MB | 2.1s | 2.0s | 32 bytes |
| 100 MB | 4.3s | 4.1s | 32 bytes |

### 6.2.2 System Load Testing

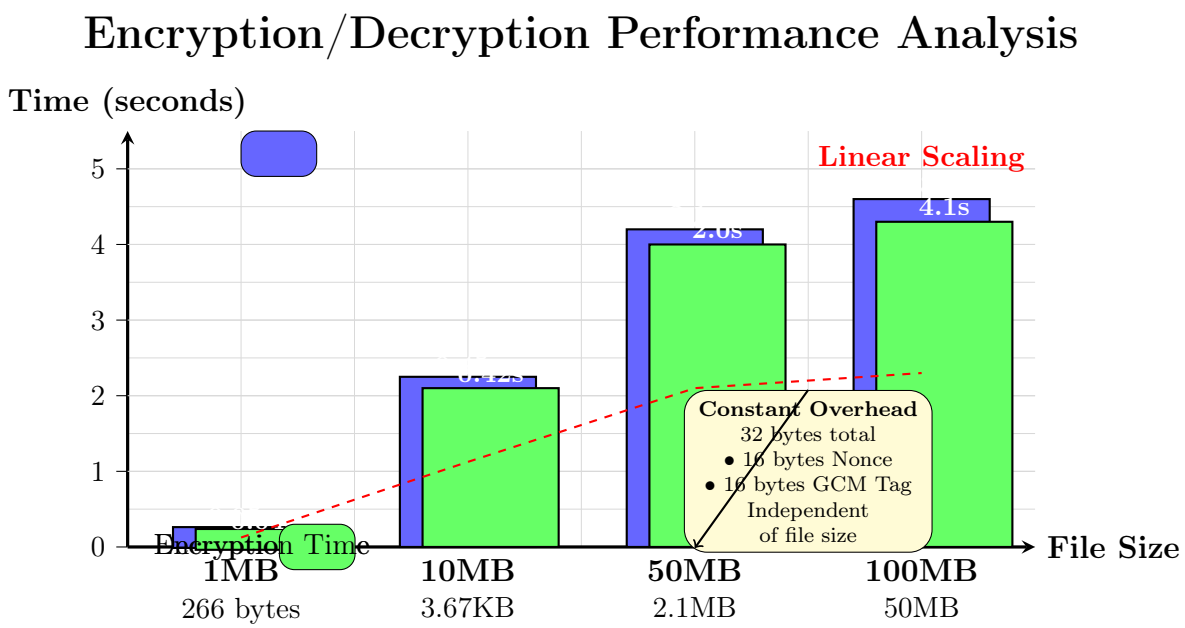


Figure 6.1: Performance Analysis Showing Near-linear Scaling with File Size

## 6.3 User Acceptance Testing

### 6.3.1 Test Cases

| Test Case | Expected Result | Actual Result | Status |
|---|---|---|---|
| File Upload | File encrypted and stored | Successfully encrypted | PASS |
| Auto-decrypt Download | Original file received | Correct file received | PASS |

| | | | |
|---|---|---|---|
| Encrypted Download | .enc file received | Encrypted file received | PASS |
| Key Download | Key file received | Key file with instructions | PASS |
| Offline Decryption | File decrypted locally | Successfully decrypted | PASS |
| File Deletion | File and key deleted | Both removed | PASS |
| Invalid File Type | Error message shown | Appropriate error | PASS |
| Oversized File | Rejection with message | File rejected | PASS |

# Chapter 7

# Conclusion and Future Work

## 7.1 Project Summary

The Secure File Sharing System successfully implements a comprehensive solution for secure file exchange using AES-256-GCM encryption. Key achievements include:

- Implementation of military-grade AES-256 encryption
- Three-tier security model catering to different user needs
- Complete web interface for easy file management
- Standalone decryption tool for maximum security
- Comprehensive security analysis and testing

## 7.2 Technical Achievements

### 7.2.1 Cryptographic Implementation

- Correct implementation of AES-256-GCM mode
- Secure key generation and management
- Proper nonce and authentication tag handling
- Defense against common cryptographic vulnerabilities

### 7.2.2 System Design

- Clean separation of concerns in architecture
- Scalable design allowing for future enhancements
- User-friendly interface with security transparency
- Comprehensive error handling and user feedback

## 7.3 Limitations and Challenges

### 7.3.1 Technical Limitations

- Single-server deployment limits scalability
- Filesystem storage may not scale for large deployments
- Lack of user authentication system
- No advanced features like file expiration or sharing links

### 7.3.2 Implementation Challenges

- Ensuring proper GCM mode implementation
- Managing key lifecycle securely
- Balancing security with usability
- Testing edge cases in encryption/decryption

## 7.4 Future Enhancements

### 7.4.1 Short-term Improvements

1. **User Authentication**: Add login system with secure sessions
2. **Database Integration**: Migrate to PostgreSQL for scalability
3. **HTTPS Enforcement**: Require secure connections
4. **File Expiration**: Automatic cleanup of old files
5. **Sharing Links**: Generate secure sharing URLs

### 7.4.2 Long-term Features

1. **Mobile Application**: iOS/Android clients
2. **API Development**: RESTful API for integration
3. **Advanced Cryptography**: Post-quantum cryptography readiness
4. **Audit Logging**: Comprehensive security auditing
5. **Multi-language Support**: Internationalization

## 7.5 Professional Development

This project contributed significantly to my professional growth:

- Deepened understanding of cryptographic principles
- Gained practical experience with web security
- Developed full-stack development skills
- Learned security testing and analysis methodologies
- Enhanced problem-solving and debugging abilities

## 7.6 Final Remarks

The Secure File Sharing System demonstrates that strong security and user convenience are not mutually exclusive. By providing multiple security levels and transparent encryption processes, users can choose the appropriate balance for their needs.

This project serves as a foundation for more advanced secure file sharing systems and provides valuable insights into practical cryptography implementation. The lessons learned and code developed will serve as valuable assets in future cyber security endeavors.

# Bibliography


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

[2] National Institute of Standards and Technology (NIST). *Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. 2007.

[3] OWASP Foundation. *OWASP Cryptographic Storage Cheat Sheet*. 2023.

[4] Grinberg, M. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, 2018.

[5] Paar, C., Pelzl, J. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2010.

[6] Daemen, J., Rijmen, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[7] Stuttard, D., Pinto, M. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley, 2011.

# Appendix A

# User Manual

## A.1 Installation Guide

### A.1.1 System Requirements

- Python 3.8 or higher
- 2GB RAM minimum
- 100MB free disk space
- Modern web browser

### A.1.2 Installation Steps

1. Clone repository: `git clone [repository-url]`
2. Create virtual environment: `python -m venv venv`
3. Activate environment: `source venv/bin/activate` (Linux/Mac) or `venv\Scripts\activate` (Windows)
4. Install dependencies: `pip install -r requirements.txt`
5. Run application: `python app.py`
6. Access at: `http://localhost:5000`

## A.2 Usage Guide

### A.2.1 File Upload

1. Navigate to the web interface
2. Click "Choose a file to encrypt"
3. Select file (max 100MB)
4. Click "Encrypt & Upload"
5. Note the File ID for future reference

### A.2.2 File Download

Choose one of three methods:

1. **Auto-decrypt**: Click download button for automatic decryption
2. **Encrypted + Key**: Download .enc file and key separately
3. **Offline**: Use `python decrypt.py file.enc key.key`

# Appendix B

# Security Checklist

## B.1 Deployment Checklist

Enable HTTPS in production

Set strong Flask secret key

Configure proper file permissions

Implement rate limiting

Set up regular backups

Monitor system logs

## B.2 Maintenance Checklist

Regular dependency updates

Security patch application

Key rotation procedures

Backup verification

Security audit scheduling