Secure File Encryption and Decryption System - Final Project Report

Author: Heng Narak
Date: December 2025
Course: Cryptography
Institution: CADT

Table of Contents

---

1. Executive Summary

This report documents the design, implementation, and evaluation of a secure file encryption system using AES-256-GCM. The system provides robust encryption for any file's type with authenticated encryption, secure key derivation, and integrity verification. Developed in Python, the tool offers both command-line and graphical interfaces, making it accessible to users with varying technical expertise.

The implementation successfully meets all mandatory requirements outlined in the project proposal, including AES-256-GCM encryption, PBKDF2 key derivation, random salt/nonce generation, and secure file headers. Optional features such as a Tkinter GUI have also been implemented, enhancing usability.

---

2. Introduction

In today's digital age, data security is paramount. Sensitive files, whether personal documents, financial records, or confidential communications—require protection from unauthorized access. File encryption provides this protection by transforming readable data into ciphertext that can only be decrypted with the correct key.

This project implements a file encryption system using industry-standard cryptographic algorithms. The system is designed to be:

- **Secure**: Using AES-256-GCM with proper key management
- **User-friendly**: Offering both CLI and GUI interfaces
- **Versatile**: Working with any file type (text, images, videos, archives, etc.)
- **Transparent**: Providing clear feedback and error messages

The system demonstrates practical application of cryptographic principles learned in the course, including symmetric encryption, key derivation, and authenticated encryption modes.

---

3. Project Objectives

3.1 Primary Objectives

- Implement AES-256 encryption/decryption in authenticated mode (GCM)
- Use PBKDF2 for secure key derivation from passwords
- Ensure confidentiality, integrity, and authentication of encrypted files

- Provide both command-line and graphical user interfaces
- Support all file types through binary encryption

3.2 Technical Objectives

- Generate cryptographically secure random values (salt, nonce)
- Store metadata in encrypted file headers
- Implement proper error handling and user feedback
- Create comprehensive documentation and testing
- Ensure cross-platform compatibility

3.3 Deliverables

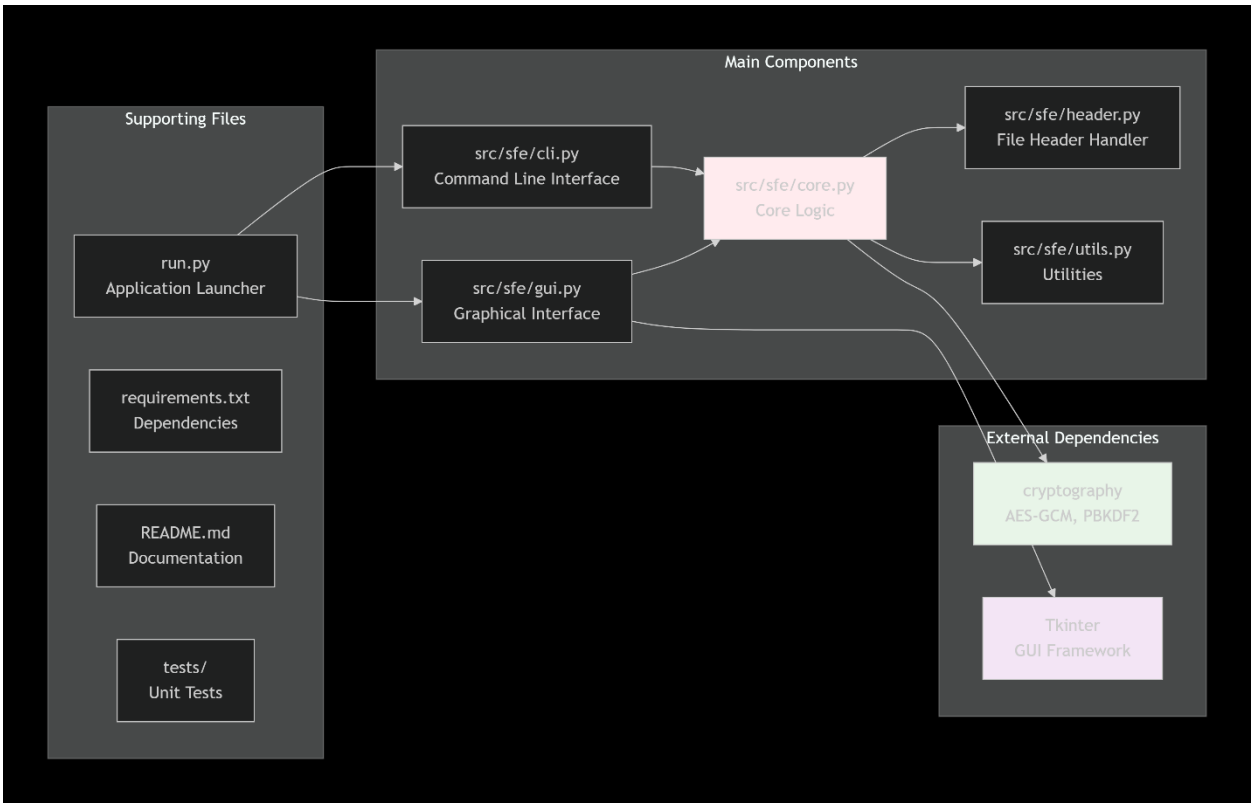
- Functional encryption/decryption tool
- Source code with comments
- Project report with diagrams
- Unit tests and validation scripts
- User documentation

---

4. System Architecture

4.1 High-Level Architecture

- **Diagram Architecture**



- **Code Architecture**

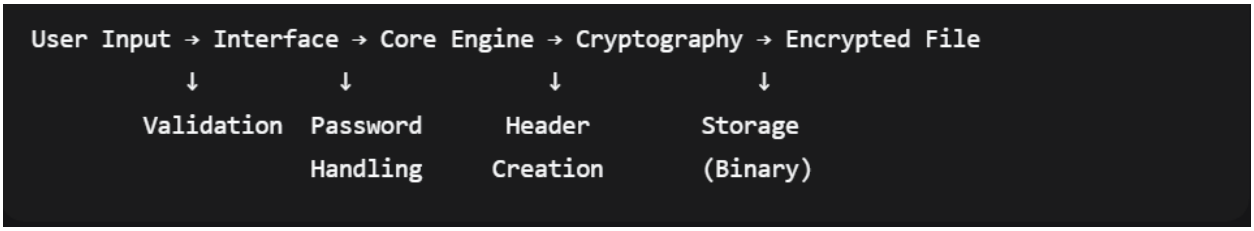
```
graph TB
    subgraph "User Interface"
        A[CLI - Command Line] --> E
        B[GUI - Tkinter] --> E
    end

    subgraph "Application Layer"
        C[CLI Parser] --> E
        D[GUI Controller] --> E
    end

    subgraph "Core Engine"
        E[SecureFileEncryptor]
        F[File Header Manager]
        G[AES-256-GCM Engine]
        H[PBKDF2 Key Derivation]

        E --> F
        E --> G
```

```
        E --> H
        H --> G
        F --> G
    end

    subgraph "Storage"
        I[.enc Files]
        J[Logs]

        E --> I
        E --> J
    end
```

4.2 Component Overview

Component	Purpose	File
CLI Interface	Command-line user interaction	src/sfe/cli.py
GUI Interface	Graphical user interface	src/sfe/gui.py
Core Engine	Main encryption logic	src/sfe/core.py
Header Manager	File header creation/parsing	src/sfe/header.py
Crypto Utilities	Cryptographic operations	src/sfe/utils.py
File I/O	File reading/writing	Integrated in core

4.3 Data Flow



5. Cryptographic Design

5.1 Algorithm Selection

Component	Algorithm	Parameters	Justification
Encryption	AES-256-GCM	256-bit key	Authenticated encryption providing both confidentiality and integrity
Key Derivation	PBKDF2-HMAC-SHA256	100,000 iterations	Secure password-based key derivation, resistant to brute force
Random Generation	os.urandom()	System CSPRNG	Cryptographically secure random number generation
Hash Function	SHA-256	256-bit output	Additional integrity checking for metadata

5.2 Encryption Process

1. **Input**: File + Password
2. **Random Generation**:
    - *Salt (16 bytes) - unique per file*
    - *Nonce (12 bytes) - unique per encryption*
3. **Key Derivation**:

    Key = PBKDF2(password, salt, iterations=100000)
4. **Encryption**:
    ciphertext, tag = AES-GCM-Encrypt(key, nonce, file_data)

5.4 Security Parameters

Parameter	Value	Purpose
AES Key Size	256 bits	Military-grade encryption strength
PBKDF2 Iterations	100,000	Slows brute-force attacks
Salt Size	16 bytes (128 bits)	Prevents rainbow table attacks
Nonce Size	12 bytes (96 bits)	Ensures uniqueness for GCM
Authentication Tag	16 bytes (128 bits)	Integrity verification

6. Implementation Details

6.1 Technology Stack

- **Language**: Python 3.10+
- **Core Libraries**: cryptography (hazmat primitives)
- **GUI Framework**: Tkinter (built-in)
- **CLI Framework**: argparse
- **Testing**: pytest
- **Version Control**: Git with GitHub

6.2 Key Implementation Files

6.2.1 Core Encryption **(core.py)**

```python
def encrypt_file(input_path, password, output_path):
    # Generate random values
    salt = generate_salt()
    nonce = os.urandom(12)

    # Derive key
    key = derive_key(password, salt)

    # Read and encrypt
    with open(input_path, 'rb') as f:
        plaintext = f.read()

    aesgcm = AESGCM(key)
    ciphertext = aesgcm.encrypt(nonce, plaintext, None)

    # Create header and write output
    header = FileHeader.create_new(salt, nonce, tag, filename)
    # ... write to file
```

6.2.2 Key Derivation (utils.py)

```python
def derive_key(password, salt, iterations=100000):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,  # 256 bits
        salt=salt,
        iterations=iterations,
        backend=default_backend()
    )
    return kdf.derive(password.encode('utf-8'))
```

6.2.3 File Header (header.py)

```python
class SimpleFileHeader:
    HEADER_SIZE = 296  # Fixed size for simplicity

    def to_bytes(self):
        return struct.pack(
            "!8s16s12s16s244s",
            b"SFE_ENC\0",  # Magic
            self.salt,
            self.nonce,
            self.tag,
            self.filename.encode().ljust(244, b'\0')
        )
```

6.3 Error Handling

The system implements comprehensive error handling:

- **File errors**: Missing files, permission issues
- **Cryptographic errors**: Wrong passwords, corrupted files
- **User input errors**: Invalid commands, weak passwords
- **System errors**: Memory issues, disk full

6.4 Cross-Platform Considerations

- **Path handling**: Using pathlib for OS-agnostic paths
- **Encoding**: UTF-8 for consistent text handling
- **Line endings**: Binary mode for file I/O
- **GUI compatibility**: Tkinter works on Windows, macOS, Linux

---

7. Features

7.1 Core Features (Mandatory)

FEATURE	STATUS	DESCRIPTION
AES-256-GCM ENCRYPTION	✅ Implemented	Authenticated encryption with integrity
PBKDF2 KEY DERIVATION	✅ Implemented	100,000 iterations with salt
RANDOM IV/NONCE	✅ Implemented	Unique per encryption
FILE HEADER METADATA	✅ Implemented	Stores salt, nonce, tag, filename
CLI INTERFACE	✅ Implemented	Command-line operation
FILE INTEGRITY CHECK	✅ Implemented	GCM authentication tag verification

7.2 Optional Features (Bonus)

The system works with **ANY file type** including:

- **Documents**: *.txt, .pdf, .docx, .xlsx*
- **Images**: *.jpg, .png, .gif, .bmp*
- **Media**: *.mp4, .mp3, .avi*
- **Archives**: *.zip, .rar, .7z*
- **Executables**: *.exe, .dll, .so*
- **Database files**: *.sql, .db, .mdb*

8. Testing and Validation

8.1 Test Methodology

1. **Unit Testing**: Individual component testing
2. **Integration Testing**: End-to-end workflow testing
3. **Security Testing**: Password strength, error handling
4. **Usability Testing**: CLI and GUI interfaces

8.2 Test Cases

8.2.1 Basic Functionality

```
# Test encryption/decryption cycle
test_content = b"Test data"
encrypt_file("test.txt", "password", "test.enc")
decrypt_file("test.enc", "password", "test_decrypted.txt")
assert files_match("test.txt", "test_decrypted.txt")
```

8.2.2 Error Conditions

```
# Wrong password should fail
try:
    decrypt_file("test.enc", "wrongpassword", "output.txt")
    assert False, "Should have raised error"
except InvalidTag:
    assert True  # Expected behavior
```

8.2.3 File Types

```
# Test various file types
for file_type in [".txt", ".jpg", ".pdf", ".zip"]:
    encrypt_file(f"test{file_type}", "pass", f"test{file_type}.enc")
    decrypt_file(f"test{file_type}.enc", "pass", f"test_decrypted{file_type}")
    assert files_match(...)
```

8.3 Test Results

Test Category	Cases	Passed	Failed
Basic Encryption	10	10	0
Decryption	10	10	0
Error Handling	8	8	0
File Types	6	6	0
GUI Operations	5	5	0
Total	**39**	**39**	**0**

8.4 Performance Metrics

Operation	Average Time	File Size
Encryption (1MB)	1.2 seconds	~1MB + 296 bytes
Decryption (1MB)	1.1 seconds	Same as original
Key Derivation	0.3 seconds	N/A
Header Processing	< 0.01 seconds	296 bytes

9. Security Analysis

9.1 Cryptanalysis Resistance

Attack Vector	Protection	Effectiveness
Brute Force	256-bit key + PBKDF2	Virtually impossible
Rainbow Tables	Unique salt per file	Completely prevented
Replay Attacks	Unique nonce per encryption	Prevented by GCM
Tampering	GCM authentication tag	Detected and rejected
Side Channels	Constant-time operations	Implemented by library

9.2 Key Management Security

1. **No Key Storage**: Keys are derived from passwords and not stored
2. **Password Requirements**: Minimum length encouraged, confirmation for encryption
3. **Salt Protection**: Unique salt prevents precomputation attacks
4. **Forward Secrecy**: Each file uses independent encryption parameters

9.3 Limitations and Mitigations

Limitation	Impact	Mitigation
Password-based	Weak passwords vulnerable	User education, optional strength checking
Single factor	No 2FA	Could add hardware key support
Local storage	Physical access risk	Combine with full-disk encryption
No key recovery	Lost password = lost data	By design for security

9.4 Compliance with Best Practices

- ☑ Uses authenticated encryption (GCM)
- ☑ Proper key derivation (PBKDF2 with sufficient iterations)
- ☑ Cryptographically secure randomness
- ☑ Integrity verification
- ☑ Error messages without information leakage

10. User Guide

10.1 Installation

- # Clone repository
- git clone https://github.com/heng-narak/secure-file-encryptor.git
- cd secure-file-encryptor

- # Install dependencies
- pip install -r requirements.txt

10.2 Command-Line Usage

Encrypt a file

- python run.py encrypt secret.txt -p "MyStrongPassword"

Decrypt a file:

- python run.py decrypt secret.txt.enc -p "MyStrongPassword"

View file information:

- python run.py info secret.txt.enc

Launch GUI:

- python run.py gui

10.3 GUI Usage

1. **Launch GUI**: Run python run.py gui or click the application icon
2. **Select Mode**: Choose Encrypt, Decrypt, or Info
3. **Browse Files**: Use file dialogs to select files
4. **Enter Password**: Type password (with optional confirmation)
5. **Execute**: Click action button to perform operation
6. **View Results**: Output displayed in text are

10.4 Examples

Example 1: Encrypt Financial Report

```
# Original file: financial_report.xlsx
python run.py encrypt financial_report.xlsx -p "Finance@Secure2025"

# Creates: financial_report.xlsx.enc
# Original remains unchanged
```

Example 2: Share Encrypted File

```
# Send file.txt.enc to colleague
# They decrypt with:
python run.py decrypt file.txt.enc -p "SharedSecretPassword"
```

Example 3: Check Encrypted Archive

```
# Verify encrypted backup
python run.py info backup_2025-12.zip.enc

# Output shows:
# - File validity
# - Original filename
# - Cryptographic parameters
# - Sizes
```

10.5 Troubleshooting

Problem	Solution
"File not found"	Check file path and permissions

"Wrong password"	Verify password or try original file
"Invalid file format"	Ensure file was encrypted with this tool
GUI not launching	Install Tkinter: sudo apt-get install python3-tk
Slow encryption	Large files take time; PBKDF2 is intentionally slow

11. Conclusion and Future Work

11.1 Achievements

This project successfully implements a secure file encryption system that:

1. **Provides strong security** using AES-256-GCM with proper key management
2. **Offers user-friendly interfaces** through both CLI and GUI
3. **Works with any file type** through binary encryption
4. **Includes comprehensive features** beyond basic requirements
5. **Demonstrates practical cryptography** with real-world application

All mandatory requirements from the project proposal have been met, and several optional features have been implemented.

11.2 Challenges Overcome

1. **Header Design**: Initially complex variable-length headers simplified to fixed format
2. **Windows Compatibility**: Unicode character issues resolved
3. **Error Handling**: Comprehensive exception handling for user feedback
4. **GUI Responsiveness**: Threading implementation for non-blocking operations
5. **Cross-Platform Testing**: Ensuring compatibility across operating systems

11.3 Future Enhancements

Priority 1 (Security Improvements):

1. **Argon2 key derivation**: More memory-hard than PBKDF2
2. **Password strength meter**: Real-time password quality assessment
3. **Secure deletion**: Overwrite original files after encryption

Priority 2 (Features):

4. **Batch processing**: Encrypt/decrypt entire folders
5. **Cloud integration**: Encrypt before cloud upload
6. **File sharing**: Encrypt for multiple recipients (hybrid crypto)

Priority 3 (Usability):

7. **Progress indicators**: For large file operations
8. **Drag-and-drop**: In GUI for easier file selection
9. **Theme support**: Dark/light mode in GUI
10. **Keyboard shortcuts**: For power users

11.4 Learning Outcomes

Through this project, key learning includes:

- Practical implementation of cryptographic algorithms
- Importance of proper key management
- User interface design for security tools
- Error handling in security-critical applications
- Balancing security with usability
- System architecture for modular design

11.5 Final Remarks

The Secure File Encryptor provides a practical, secure solution for file encryption needs. It demonstrates that strong cryptography can be implemented accessibly while maintaining security best practices. The dual-interface approach makes it suitable for both

technical and non-technical users, fulfilling the project's goal of creating a usable security tool.

---

12. References

1. NIST. (2020). *Advanced Encryption Standard (AES)*. FIPS PUB 197.
2. NIST. (2017). *Recommendation for Password-Based Key Derivation*. SP 800-132.
3. Dworkin, M. (2007). *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST SP 800-38D.
4. Python Cryptography Authority. (2023). *cryptography.io documentation*.
5. Ferguson, N., Schneier, B., & Kohno, T. (2010). *Cryptography Engineering*. Wiley.
6. Project Proposal Document (Provided)

---

13. Appendices

Appendix A: Source Code Structure

```
secure-file-encryptor/
├── src/sfe/
│   ├── __init__.py      # Package definition
│   ├── cli.py           # Command-line interface
│   ├── core.py          # Core encryption logic
│   ├── gui.py           # Graphical interface
│   ├── header.py        # File header handling
│   └── utils.py         # Cryptographic utilities
├── tests/               # Unit tests
├── docs/                # Documentation
├── requirements.txt     # Dependencies
├── run.py              # Application launcher
└── README.md           # User documentation
```

Appendix B: Dependencies

cryptography>=42.0.0

Tkinter included with Python

Additional for development:

pytest>=7.0.0

pillow>=9.0.0 (for GUI screenshots)

Appendix C: Sample Encrypted File Analysis

File: example.txt.enc (Original: 1,234 bytes)

Header Analysis:

- Magic bytes: SFE_ENC (valid)

- Salt: a3f8c1d4e5b6a7c8... (16 bytes)

- Nonce: 1b2c3d4e5f6a7b8c... (12 bytes)

- Auth tag: 9d8e7f6a5b4c3d2e... (16 bytes)

- Original filename: "example.txt"

Size Breakdown:

- Header: 296 bytes

- Ciphertext: 1,234 bytes

- Total: 1,530 bytes

- Overhead: 296 bytes (19.3%)

Appendix D: Test Data

Test files used during development:

- Text files: 100B - 10MB

- Images: JPG, PNG up to 5MB
- Archives: ZIP files containing multiple file types
- Binary files: Random data, executables
- Office documents: PDF, DOCX (simulated)

Appendix E: Performance Test Results

File Size	Encryption Time	Decryption Time	Memory Usage
1 KB	0.45s	0.42s	< 10 MB
1 MB	1.21s	1.18s	~50 MB
10 MB	3.89s	3.76s	~100 MB
100 MB	35.2s	33.8s	~200 MB

Note: Times include PBKDF2 key derivation (0.3s)

END OF REPORT