File Encoder Decoder Tool

Operating Systems Course Project

Agrapujya Lashkari, Kartik Yadav
Computer Science and Artificial Intelligence
Rishihood University

***Abstract*—This paper presents the design and implementation of a web-based file encryption and decryption system that demonstrates key Operating Systems concepts including file system operations, process isolation, memory management, and security primitives. The system implements two encryption modes: password-based symmetric encryption using AES-256-GCM with PBKDF2 key derivation, and public-key hybrid encryption using RSA-OAEP combined with AES-256-GCM. The application is built using Next.js with both client-side (WebCrypto API) and server-side (Node.js crypto) implementations, showcasing the differences between user-space browser sandboxing and server-side process execution. Custom binary (.osenc) and JSON-based (.osencpk) file formats are designed to store encrypted data along with necessary cryptographic parameters.**

***Index Terms*—file encryption, AES-256-GCM, RSA-OAEP, hybrid encryption, PBKDF2, WebCrypto, operating systems, key management**

I. Introduction

File encryption is a fundamental security mechanism that protects sensitive data from unauthorized access. This project implements a comprehensive file encryption system that serves dual purposes: providing practical file security functionality while demonstrating core Operating Systems concepts.

The system addresses two distinct use cases:

1) **Password-based encryption**: Where the sender and recipient share a common password used to derive encryption keys.
2) **Public-key encryption**: Where files can be encrypted for specific recipients without sharing any secrets, using asymmetric cryptography.

A. Operating Systems Relevance

This project demonstrates several OS concepts:

- **File Systems**: Browser File API vs. native system calls
- **Process Isolation**: Browser sandbox vs. serverless function containers
- **Memory Management**: Buffer handling and garbage collection
- **Security**: Cryptographic primitives, key management, trust boundaries

II. System Architecture

A. Technology Stack

The application is built using:

- **Frontend**: Next.js 16 with React and TypeScript
- **Styling**: Tailwind CSS
- **Client Crypto**: WebCrypto API (browser-native)
- **Server Crypto**: Node.js crypto module
- **Deployment**: Vercel serverless platform

B. Execution Modes

The system supports two execution modes:

1) Local (Browser) Mode: All cryptographic operations execute within the browser's JavaScript runtime using the WebCrypto API. Files never leave the user's device, providing maximum privacy. This mode:

- Uses the browser's sandboxed execution environment
- Accesses files through the File API (user-consent based)
- Stores keys in browser localStorage

2) Server Mode: Files are uploaded to serverless API routes where Node.js performs encryption/decryption. This mode:

- Runs in isolated containers on Vercel
- Has file size limits due to serverless constraints (4.5MB)
- Only supports password-based encryption (no access to client keys)

III. Cryptographic Design

A. Password-Based Encryption (.osenc)

1) Key Derivation Function: The system uses PBKDF2 (Password-Based Key Derivation Function 2) with the following parameters:

- Hash function: SHA-256
- Iterations: 310,000 (OWASP recommended minimum)
- Salt: 16 bytes (randomly generated per file)
- Output: 256-bit key

The mathematical formulation of PBKDF2 is:

$$DK = T_1 \| T_2 \| ... \| T_{dkLen/hLen} \tag{1}$$

where each block T_i is computed as:

$$T_i = F(Password, Salt, c, i) \tag{2}$$

$$F(P, S, c, i) = U_1 \oplus U_2 \oplus ... \oplus U_c \tag{3}$$

with $U_1 = PRF(P, S \| INT(i))$ and $U_j = PRF(P, U_{j-1})$ for $j > 1$.

2) Encryption Algorithm: AES-256-GCM (Advanced Encryption Standard with Galois/Counter Mode) provides both confidentiality and authenticity:

- Key size: 256 bits
- IV (Initialization Vector): 12 bytes (96 bits), randomly generated
- Authentication tag: 16 bytes (128 bits)

GCM mode combines CTR (Counter) mode encryption with GHASH authentication:

$$C_i = E_K(Counter_i) \oplus P_i \tag{4}$$

$$Tag = GHASH_H(A \| C) \oplus E_K(Counter_0) \tag{5}$$

B. Public-Key Encryption (.osencpk)

1) Hybrid Encryption Approach: Pure RSA encryption is limited by message size (approximately key size minus padding). For encrypting arbitrary files, we use hybrid encryption:

1) Generate a random 256-bit AES session key
2) Encrypt the file with AES-256-GCM using the session key
3) Encrypt the session key with recipient's RSA public key
4) Package both encrypted components together

2) RSA-OAEP: RSA with Optimal Asymmetric Encryption Padding:

- Key size: 4096 bits
- Hash: SHA-256
- Public exponent: 65537 ($2^{16} + 1$)

RSA encryption is based on the difficulty of factoring large integers:

$$n = p \times q \tag{6}$$

$$\phi(n) = (p-1)(q-1) \tag{7}$$

$$e \cdot d \equiv 1 \pmod{\phi(n)} \tag{8}$$

Encryption: $C = M^e \mod n$
Decryption: $M = C^d \mod n$
OAEP padding prevents various attacks on textbook RSA by adding randomness and structure to the message before encryption.

3) Multi-Recipient Support: The system supports encrypting for multiple recipients:

- A single AES session key encrypts the file once
- The session key is wrapped separately for each recipient's public key
- Any recipient can decrypt using their private key
- File size increases linearly with number of recipients (512 bytes per recipient for RSA-4096)

IV. File Format Specifications

A. .osenc Format (Version 1)

Binary format with the following structure:

TABLE I
.OSENC BINARY LAYOUT

Field	Size	Description
Magic	6 bytes	ASCII "OSENC1"
Version	1 byte	0x01
Iterations	4 bytes	PBKDF2 iterations (BE)
Original Size	4 bytes	Original file size (BE)
Salt Length	1 byte	Length of salt
IV Length	1 byte	Length of IV
Filename Length	2 bytes	UTF-8 filename length
MIME Length	2 bytes	MIME type length
Salt	variable	Random salt
IV	variable	Random IV
Filename	variable	Original filename
MIME	variable	MIME type
Ciphertext	variable	Encrypted data + tag

B. .osencpk Format (Version 2)

JSON-based format for public-key encryption:

```
{
  "v": 2,
  "iv": "<base64>",
  "wrappedKeys": [
    {
      "label": "Recipient Name",
      "wrappedKey": "<base64>"
    }
  ],
  "ciphertext": "<base64>",
  "filename": "original.txt",
  "mime": "text/plain",
  "originalSize": 12345
}
```

V. Key Management

A. Key Generation

RSA-4096 keypairs are generated using WebCrypto's `generateKey` function with:

- Algorithm: RSA-OAEP
- Modulus length: 4096 bits
- Public exponent: [0x01, 0x00, 0x01] (65537)
- Hash: SHA-256

B. Key Storage

Keys are stored in browser localStorage:

- Public key: SPKI format, base64 encoded
- Private key: JWK (JSON Web Key) format

C. Key Export/Import

Users can export their keypair as a JSON file for backup:

```
{
  "version": 1,
  "exportedAt": "2026-02-25T...",
  "publicKey": "<base64 SPKI>",
  "privateKey": { /* JWK */ }
}
```

VI. Security Analysis

A. Threat Model

The system protects against:

- Unauthorized file access (encryption)
- Data tampering (GCM authentication)
- Password brute-force (high PBKDF2 iterations)
- Man-in-the-middle (public-key mode)

B. Security Properties

- **Confidentiality**: AES-256 provides 256-bit security
- **Integrity**: GCM tag detects any modification
- **Authenticity**: Decryption fails if tampered
- **Forward secrecy**: Each file has unique salt/IV

VII. OS Concepts Demonstrated

A. File System Operations

- Browser: Sandboxed File API with user consent
- Server: Ephemeral in-memory processing
- No direct disk access in either mode

B. Process Isolation

- Browser tab: Isolated JavaScript context
- Serverless: Container-based isolation
- Private keys never cross trust boundaries

C. Memory Management

- Large files cause memory pressure
- Server mode limits file size
- Garbage collection handles buffer cleanup

VIII. Implementation Details

A. Encryption Flow (Password Mode)

$salt \leftarrow randomBytes(16)$
$iv \leftarrow randomBytes(12)$
$key \leftarrow PBKDF2(password, salt, 310000)$
$ciphertext \leftarrow AES\text{-}GCM\text{-}Encrypt(key, iv, plaintext)$
$output \leftarrow encode\_osenc(salt, iv, ciphertext, metadata)$

B. Encryption Flow (Public-Key Mode)

$aesKey \leftarrow generateRandomKey(256)$
$iv \leftarrow randomBytes(12)$
$ciphertext \leftarrow AES\text{-}GCM\text{-}Encrypt(aesKey, iv, plaintext)$

for each $recipient$ in $recipients$ **do**
  $wrappedKey \leftarrow RSA\text{-}OAEP\text{-}Encrypt(recipient.publicKey, aesKey)$
end for
$output \leftarrow encode\_osencpk(iv, wrappedKeys, ciphertext, metadata)$

IX. Results and Discussion

The implemented system successfully demonstrates:

1) Secure file encryption with industry-standard algorithms
2) Two distinct encryption paradigms (symmetric and hybrid)
3) Clear separation between browser and server execution
4) Practical key management for end users

A. Limitations

- Large file handling limited by memory
- No streaming encryption support
- Browser localStorage has size limits
- Server mode unavailable for public-key encryption

X. Conclusion

This project successfully implements a dual-mode file encryption system that serves both practical security needs and educational purposes for Operating Systems concepts. The password-based mode offers simplicity with shared secrets, while the public-key mode enables secure file sharing without prior secret exchange. The system demonstrates how modern web applications can leverage browser sandboxing and serverless architectures while maintaining strong cryptographic security.

XI. Future Work

- Implement streaming encryption for large files
- Add digital signatures for sender authentication
- Support hardware security keys (WebAuthn)
- Implement secure key exchange protocols

References

[1] NIST, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," SP 800-38D, 2007.
[2] IETF, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, 2017.
[3] RSA Laboratories, "RSA Cryptography Standard," PKCS #1 v2.2, 2012.
[4] W3C, "Web Cryptography API," W3C Recommendation, 2017.
[5] OWASP, "Password Storage Cheat Sheet," 2024.
[6] A. Silberschatz, P. Galvin, G. Gagne, "Operating System Concepts," 10th ed., Wiley, 2018.