SfEncryptor Documentation

Chapter 1: Introduction

- 1.1. Overview of SfEncryptor
- 1.2. Purpose and Goals
- 1.3. Key Features

Chapter 2: Getting Started

- 2.1. Installation
- 2.2. Dependencies
- 2.3. Basic Usage Example

Chapter 3: Core Functionality

- 3.1. Encryption Methods
- 3.2. Decryption Methods
- 3.3. Key Management

Chapter 4: Advanced Usage

- 4.1. Customization Options
- 4.2. Error Handling
- 4.3. Best Practices

Chapter 5: Contributing

- 5.1. Contribution Guidelines
- 5.2. Reporting Issues
- 5.3. Code Style

Chapter 1: Introduction

1.1 Overview of SfEncryptor

SfEncryptor is a robust, modern desktop application designed to securely encrypt and decrypt files across multiple platforms including Windows, macOS, and Linux. Built with PyQt6, it offers a polished graphical user interface that makes cryptography accessible to both technical users and those less familiar with encryption concepts. At its core, SfEncryptor employs the Python Cryptography library, focusing on strong, industry-standard cryptographic primitives—defaulting to AES-256 in Galois/Counter Mode (GCM), which ensures both confidentiality and data integrity.

Key highlights include:

- **User-friendly UI:** Drag-and-drop functionality, intuitive key management, and a clear workflow reduce user error.
- **Security-first design:** AES-256-GCM provides authenticated encryption, guaranteeing that ciphertext hasn't been tampered with.
- **Extensibility:** A plugin system allows developers to add or customize encryption algorithms without modifying core code.
- **Command-line interface:** Enables scripting, automation, and headless operation.
- **Secure deletion:** Optional multi-pass file overwriting minimizes forensic recoverability of original plaintext files.
- **Metadata sidecar files:** These files store encryption parameters such as nonce and algorithm identifiers separately but alongside the encrypted files, facilitating seamless decryption without user guesswork.

1.2 Purpose and Goals

SfEncryptor's mission is to democratize strong encryption by making it approachable and reliable across platforms, providing users with:

- **Accessibility:** Simple workflows that hide complex cryptography behind a clean interface.
- **Cross-platform consistency:** Ensuring the same encrypted files can be decrypted on any supported OS.
- **Key lifecycle safety:** Offering users straightforward, safe ways to generate, import, export, label, and delete keys, minimizing risks of key leakage or loss.
- **Reduced user configuration errors:** By embedding critical parameters like nonce and algorithm details into metadata files, the app prevents configuration mistakes that could otherwise cause data loss.
- **Future-proof architecture:** The plugin system anticipates future cryptographic needs and new algorithms, enabling modular growth.
- **Secure destruction:** For sensitive workflows, securely overwriting original plaintext files reduces residual data risk.

1.3 Key Features

Feature	Description
Modern UI	Built on PyQt6 for a responsive, intuitive GUI
Cross-platform	Works seamlessly on Windows, macOS, and Linux
AES-256-GCM	Default encryption mode providing confidentiality and integrity
Drag & Drop	Easily add files/folders by dragging into the window
Secure Deletion	Multi-pass overwrite option for safe file removal
Metadata Sidecar Files	Automatic creation and management of metadata for encryption parameters
Key Management Tab	Generate, import, export (with optional password protection), and delete keys; assign human-readable labels
Plugin System	Add or customize encryption algorithms via plugin modules
Command-line Interface	Full CLI support for scripting and automation
Theming	Support for light/dark modes and modern styling

Chapter 2: Getting Started

2.1 Installation

Prerequisites:

- Python 3.7 or higher installed on your system.

Step-by-step Installation:

1. **Clone the repository:**
```
git clone https://github.com/surya-sx/SfEncryptor.git
cd SfEncryptor
```
4. **(Optional) Create and activate a Python virtual environment:**
   This isolates dependencies, ensuring they don't conflict with other Python projects.
   - On Windows:
   - `python -m venv .venv`
   - `.venv\Scripts\activate`
   - On macOS/Linux:
   - `python -m venv .venv`
   - `source .venv/bin/activate`
5. **Install required dependencies:**
```
pip install pyqt6 cryptography
```

   Or if the project contains a `requirements.txt` file:

```
pip install -r requirements.txt
```

7. **Run the application:**
```
python SfEncryptor.py
```

2.2 Dependencies

- **Python 3.7+** — Core runtime environment.
- **PyQt6** — Provides the graphical user interface components, enabling native-looking windows, dialogs, and widgets.
- **cryptography library** — Implements AES-256-GCM and other cryptographic primitives.
- **Standard Python libraries** — Used for file system operations, path handling, logging, and subprocess management.

2.3 Basic Usage Example

GUI Workflow

- Launch the app with `python SfEncryptor.py`.
- Drag and drop files or entire folders into the main window.
- Select encryption algorithm and parameters in the settings (AES-256-GCM is default and recommended).
- Navigate to the Key Management tab to generate a new key or import an existing key.
- Click "Encrypt" to secure files, or "Decrypt" for files already encrypted.
- Optionally, enable "secure deletion" to overwrite the original plaintext files after successful encryption.
- The encrypted files will be saved alongside metadata sidecar files containing the encryption parameters.

Example CLI Usage

(Assuming CLI flags implemented as described)

- Encrypt a file:
- `python SfEncryptor.py --encrypt --input /path/to/file.txt --key /path/to/keyfile --out /path/to/file.enc`
- Decrypt a file:
- `python SfEncryptor.py --decrypt --input /path/to/file.enc --key /path/to/keyfile --out /path/to/file.txt`
- Display help:
- `python SfEncryptor.py --help`

Chapter 3: Core Functionality

3.1 Encryption Methods

Default: AES-256-GCM
AES (Advanced Encryption Standard) with a 256-bit key size is an industry gold standard for symmetric encryption. Using Galois/Counter Mode (GCM) ensures both:

- **Confidentiality:** Data is encrypted so unauthorized users cannot read it.
- **Integrity:** Authentication tags verify that the ciphertext and associated data have not been tampered with.

Detailed Steps:

1. **Key acquisition:**
   The application obtains a 256-bit symmetric key from the key management system, either by generating a new one or loading an existing key.
2. **Nonce/IV generation:**
   Each encryption operation uses a unique nonce (Initialization Vector), essential to ensure GCM's security guarantees.
3. **Encryption:**
   The file is encrypted in chunks (supporting large files without loading all into memory). The process generates ciphertext plus an authentication tag.
4. **Metadata creation:**
   Nonce, algorithm identifiers, tag locations, and other parameters are saved in a metadata sidecar file next to the encrypted file.
5. **Optional secure deletion:**
   If enabled, the original plaintext file is overwritten multiple times with random data to prevent forensic recovery.

Plugin Support:
Developers can add new algorithms or workflows by placing plugin modules into the `plugins/` directory. Plugins must implement a standard interface handling encryption, decryption, and metadata serialization.

3.2 Decryption Methods

- Metadata sidecar files are read to determine which algorithm, nonce, and parameters were used.
- The correct key is loaded and verified for length and format.
- The ciphertext is decrypted with authentication tag verification.
- If the tag verification fails, indicating corruption or tampering, decryption is aborted with an error; no partial data is output.

3.3 Key Management

- **Generate keys:** Create secure, random 256-bit symmetric keys.
- **Import keys:** Load keys from files or clipboard input.
- **Export keys:** Save keys to files, optionally encrypting the key file with a password for added security.
- **Delete keys:** Remove references and optionally securely erase key files.
- **Label keys:** Assign human-readable labels to keys to organize multiple keys easily.

Security Note:
The application ensures keys are never written to logs or temporary files in plaintext and aims to minimize their time in memory.

Chapter 4: Advanced Usage

4.1 Customization Options

- **Plugins:**
  Add custom encryption algorithms or processing workflows by implementing the

expected interface (`encrypt()`, `decrypt()`, and metadata handlers). Drop these modules into `plugins/`.

- **Theming:**
  Modify the UI appearance using Qt style sheets, enabling dark mode, high contrast, or user-preferred color schemes.
- **Metadata Extensions:**
  Extend the metadata format to include additional fields such as the hash of the original filename, compression flags, or timestamps to improve data tracking.
- **Batch Automation:**
  Use the CLI to encrypt or decrypt entire directories in batch scripts for automated workflows.
- **Secure Deletion Passes:**
  Configure the number of overwrite passes (more passes increase security but take longer).

4.2 Error Handling

- **Key errors:**
  Detect invalid or missing keys early and inform the user.
- **Integrity failures:**
  Immediately halt on authentication tag mismatches and notify the user to prevent corrupted data usage.
- **File I/O errors:**
  Handle permission issues, disk space problems, or locked files gracefully with clear messages.
- **Plugin loading errors:**
  Detect missing dependencies or interface mismatches and log detailed info for developers.
- **Metadata corruption:**
  Validate sidecar files and refuse to decrypt if they're missing or malformed.

Best Practices:
Fail fast and fail safely with clear, user-friendly error messages without exposing internal cryptographic details.

4.3 Best Practices

- **Nonce uniqueness:**
  Never reuse a nonce with the same key in AES-GCM — the app enforces this automatically.
- **Key backups:**
  Store keys securely, preferably offline or in hardware modules.
- **Password protection:**
  Password-encrypt exported key files to mitigate key theft risk.
- **Dependencies:**
  Keep PyQt6, cryptography, and other libraries up to date to benefit from security patches.
- **Data validation:**
  Verify decrypted data integrity beyond GCM if needed (e.g., using hashes).

- **Trusted plugins only:**
  Since plugins have access to plaintext data, install only plugins from trusted sources.
- **Use secure deletion cautiously:**
  Only enable when original plaintext files are confirmed no longer needed.
- **Least privilege execution:**
  Run SfEncryptor with minimal permissions to reduce security risks.

Chapter 5: Contributing

5.1 Contribution Guidelines

- **Fork the repo** and create feature branches with descriptive names (e.g., `feature/add-chacha20`).
- Write clear, concise commit messages indicating the change purpose.
- When submitting Pull Requests, include:
  - Summary of the change
  - Motivation for the feature/fix
  - How to test it
  - Screenshots if the UI is affected
- Update documentation accordingly.
- Add or suggest automated tests if possible.

5.2 Reporting Issues

Include:

- Clear description and steps to reproduce the issue
- Expected vs actual behavior
- Environment info: OS, Python version
- Relevant logs or screenshots (ensuring no sensitive info)
- For security bugs, follow the responsible disclosure outlined in `SECURITY.md`.

5.3 Code Style

- Follow PEP 8 for readability.
- Use `snake_case` for functions and variables; `PascalCase` for class names.
- Centralize cryptographic constants and parameters.
- Separate UI code from cryptographic logic.
- Provide detailed docstrings on all public functions and classes explaining parameters, return values, and exceptions.
- Maintain a minimal and clearly documented plugin interface to encourage community contributions.

Appendix: Future Enhancements (Ideas)

- Support for additional algorithms like ChaCha20-Poly1305, RSA+AES hybrid encryption.
- Adding compression before encryption to reduce file sizes.

- Metadata versioning for backward compatibility and enhanced integrity checks.
- Automated key rotation and expiration management.
- Headless service mode for scheduled, unattended encryption/decryption tasks.

Licensing and Contact

- **License:** MIT License (see LICENSE file)
- **Security Policy:** See SECURITY.md for responsible disclosure
- **Author Contact:** Surya B — myselsuryaaz@gmail.com | GitHub Profile