International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 07 | July - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

Secure File Encryption using AES with Password Protection

Praveen Hebbal[1], Gopinath Ramaje[1], Prajwal Biradar[1], Pavan Kumar R[1], Prof. Deepika Dash[*]
[1]BE students, Dept. of Computer Science and Engineering, R V College of Engineering
*Assistant Professor, Dept. of Computer Science and Engineering, R V College of Engineering

***Abstract*—In today's digital age, the importance of data confidentiality and integrity cannot be overstated. With the ever-growing reliance on cloud storage, file sharing, and personal computing, protecting sensitive information from unauthorized access is critical. This paper introduces a file encryption system that leverages the Advanced Encryption Standard (AES) algorithm combined with password-based authentication to safeguard user files. Developed using Python, the tool offers a graphical user interface (GUI) for easy interaction, enabling users to encrypt or decrypt files using a user-specified password. The system hashes the password to derive a secure encryption key, ensuring that the key is never stored or transmitted in plaintext. Through extensive testing, the application demonstrates robust performance across a range of file types and sizes, offering a secure yet accessible solution for everyday data protection needs.**

***Index Terms*—Cryptography, AES, File Encryption, Python, Password Protection, Data Security, GUI.**

I. Introduction

In today's digital landscape, data breaches, unauthorized access, and identity theft have become common threats to personal and organizational information. With files being regularly stored on local drives, transferred over networks, or uploaded to cloud services, there is an urgent need to secure them from unauthorized access. Traditional file protection methods like hidden folders or operating system-level access control are often insufficient against skilled adversaries or malicious software. Consequently, cryptographic techniques—especially encryption—are now widely adopted for securing digital data.

Encryption converts readable data (plaintext) into unreadable ciphertext, ensuring that only those possessing the correct key can recover the original content. Among various cryptographic algorithms, the Advanced Encryption Standard (AES) is globally accepted for its speed and strength. It is a symmetric block cipher, adopted by the U.S. National Institute of Standards and Technology (NIST), and is used across various domains including finance, healthcare, defense, and cloud computing. AES-256, the strongest variant, uses a 256-bit key and provides high security with reasonable performance.

However, many encryption tools in the market are either commercial, complicated, or require high technical expertise to operate. This creates a barrier for individuals or small organizations that need quick and reliable file-level encryption without deep cryptographic knowledge.

To address this, the proposed project presents a Python-based file encryption system that uses AES-256 encryption combined with password-derived keys through SHA-256 hashing. The password is never stored and acts as a secure mechanism for key generation. Furthermore, a Graphical User Interface (GUI) using Tkinter simplifies the user experience, enabling file selection, encryption, and decryption with minimal effort.

This paper presents the system design, methodology, cryptographic implementation, and experimental evaluation of the application. The solution serves as a lightweight, open-source, and secure tool that can be used for protecting personal or academic files across Windows and Linux platforms.

II. Related Work

File encryption has long been a core strategy in ensuring data confidentiality, particularly in systems exposed to potential security breaches, theft, or unauthorized access. Various tools and libraries—both commercial and open-source—have emerged to address the need for secure file storage, each offering different trade-offs in terms of usability, security, and integration.

A. Commercial Encryption Tools

Several commercial-grade tools provide end-to-end file encryption solutions. Notable among them is AxCrypt, which offers AES-based encryption with cloud storage integration. It supports both individual and team-based file sharing but locks advanced features behind paid tiers. Similarly, BitLocker, integrated into the Windows operating system, provides full-disk encryption but lacks portability across non-Windows platforms and requires system-level permissions for configuration.

Another example is VeraCrypt, an open-source successor to TrueCrypt, which allows users to create encrypted file containers or encrypt entire volumes. While it offers robust protection, the tool is often seen as too complex for casual users, requiring significant configuration and familiarity with disk volumes and mount points.

Despite the strength of these tools, their complexity or pricing structures limit accessibility—especially for students, personal users, or lightweight scenarios where only individual files need encryption.

B. Open-Source File Encryption Tools

Open-source file encryption tools such as GnuPG and Cryptomator offer a high degree of transparency and customization. GnuPG (GPG) is a widely used implementation of the

OpenPGP standard and supports asymmetric encryption using RSA and ECC algorithms. However, it is primarily command-line based and intended for advanced users familiar with key management concepts.

C. Password-Based Key Derivation and AES in Practice

The concept of password-derived encryption keys is commonly used to bridge the gap between user-friendliness and cryptographic strength. Instead of managing complex keys, users can rely on familiar passwords which are transformed into cryptographically secure keys using hashing or Key Derivation Functions (KDFs). Among these, SHA-256 is widely accepted for its balance between performance and security.

AES (Advanced Encryption Standard), particularly AES-256, is a symmetric cipher known for resisting brute-force attacks due to its large key space. It is extensively used in encrypted messaging, cloud storage, and secured backups.

Academic studies and practical implementations have validated AES's resilience against known cryptanalytic methods. For example, the National Security Agency (NSA) has approved AES-256 for encrypting classified documents, underlining its relevance for both personal and governmental security needs.

These tools, while powerful, do not often cater to end-users looking for simple file-level encryption with password access and an intuitive interface.

D. Gaps and Motivation

Despite the wide availability of encryption utilities, a significant gap remains in the form of lightweight tools designed specifically for individual file protection with simplicity and accessibility in mind. Most existing solutions are geared toward full-disk encryption or enterprise-level use, which often brings unnecessary complexity for users with basic encryption needs. These tools frequently require internet connectivity, account registration, or integration with cloud services, making them unsuitable for users who prefer offline control or minimal setup. Moreover, while password-based encryption is a common feature, many applications store or manage encryption keys in a way that may introduce security risks if not properly handled.

In contrast, the proposed system addresses these shortcomings by focusing on local file-level encryption with a self-contained architecture. It uses AES-256 for robust data security and derives encryption keys directly from user-provided passwords using SHA-256 hashing, without storing any passwords or keys. The application is implemented entirely in Python and features a graphical user interface built with Tkinter, making it approachable for users with little to no technical background. No internet access or user accounts are required, and the tool operates entirely offline, preserving user autonomy and privacy. By combining strong encryption practices with simplicity and portability, this system is particularly well-suited for students, educators, and small-scale users seeking a secure yet user-friendly encryption solution.

III. Methodology

The architecture of the application consists of four primary layers: the user interface layer, the password handling and key derivation layer, the encryption engine, and the file input/output (I/O) handler. These layers interact in a pipeline manner, enabling users to encrypt or decrypt files with minimal interaction.

The user initiates the process by selecting a file through the graphical interface and entering a password. This password is never stored or transmitted. Instead, it is converted into a fixed-length 256-bit key using the SHA-256 hash function. The resulting key is then passed to the AES encryption module, which encrypts the file in Cipher Block Chaining (CBC) mode using a randomly generated Initialization Vector (IV). The IV is stored at the beginning of the encrypted file to facilitate decryption. The file is then saved with a modified extension indicating its encrypted state.

For decryption, the process is reversed. The system reads the encrypted file, extracts the IV, and derives the encryption key from the password using the same hashing process. If the key is valid and the password matches, the file is successfully decrypted and restored to its original form.

A. System Architecture

The architecture of the application consists of four primary layers: the user interface layer, the password handling and key derivation layer, the encryption engine, and the file input/output (I/O) handler. These layers interact in a pipeline manner, enabling users to encrypt or decrypt files with minimal interaction.

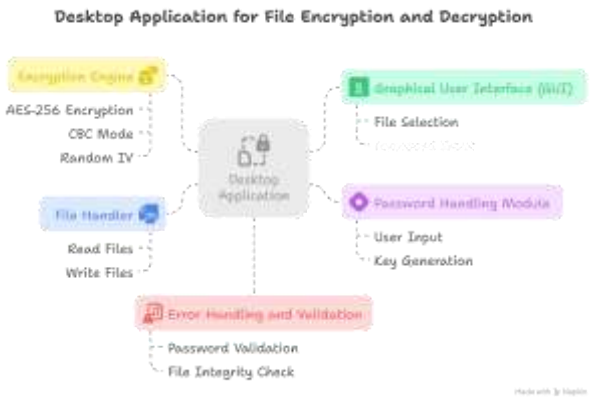

Fig. 1. System Architecture Flowchart of the Encryption Application

The user initiates the process by selecting a file through the graphical interface and entering a password. This password is never stored or transmitted. Instead, it is converted into a fixed-length 256-bit key using the SHA-256 hash function. The resulting key is then passed to the AES encryption module, which encrypts the file in Cipher Block Chaining (CBC) mode using a randomly generated Initialization Vector (IV). The IV is stored at the beginning of the encrypted file to facilitate

decryption. The file is then saved with a modified extension indicating its encrypted state.

For decryption, the process is reversed. The system reads the encrypted file, extracts the IV, and derives the encryption key from the password using the same hashing process. If the key is valid and the password matches, the file is successfully decrypted and restored to its original form.

B. Password Handling and Key Derivation

To eliminate the need for explicit key management, the system employs a password-based encryption approach. When a user inputs a password, it is processed through the SHA-256 hashing algorithm, producing a 256-bit cryptographic key. This method ensures that the key has sufficient entropy and uniformity. As a one-way hash function, SHA-256 offers resistance against brute-force and rainbow table attacks, enhancing the security of the derived key.

Unlike simple password storage mechanisms, the system never saves the password or the key on disk. This ensures that even in the event of system compromise, sensitive key material remains inaccessible. Additionally, the use of hashing for key derivation ensures deterministic key generation, allowing the same password to consistently produce the same encryption key.

C. AES Encryption and Decryption Process

The encryption engine utilizes the Advanced Encryption Standard (AES) with a 256-bit key in CBC mode. AES is a symmetric-key algorithm that operates on fixed-size blocks of plaintext.

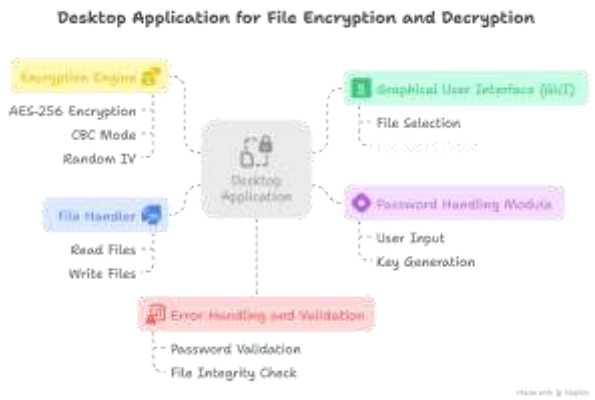

Fig. 2. System Architecture Flowchart of the Encryption Application

In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted, which prevents identical plaintext blocks from producing identical ciphertext blocks.

A 16-byte Initialization Vector (IV) is randomly generated for each encryption session. This IV is crucial in ensuring that identical files encrypted with the same password result in different ciphertexts. The IV is stored along with the encrypted content and used during the decryption process. Padding is applied to ensure that the plaintext length is a multiple of the AES block size (16 bytes).

For decryption, the system reads the stored IV, re-derives the key from the user's password, and attempts to reverse the encryption process. If the password is incorrect, the decryption fails, and the user is notified. This prevents unauthorized access and ensures that only those with the correct password can retrieve the original file content.

D. Graphical User Interface (GUI)

To ensure accessibility for users without programming or cyber security knowledge, the system includes a GUI developed using Python's tkinter library. The interface provides buttons for selecting files, entering passwords, and performing encryption or decryption operations. Status messages and prompts guide the user through each step, minimizing the potential for errors.

The GUI abstracts the underlying cryptographic operations and file management processes, making the application suitable for educational institutions, casual users, or small organizations. The design emphasizes simplicity, with clearly labeled controls and minimal dependencies.

E. File Handling and Output Management

Encrypted files are saved with a new extension (e.g., .enc) to indicate their encrypted status. During encryption, original files are preserved by default unless the user opts for overwriting. The system ensures that temporary files or partial outputs are securely handled and removed in the event of an error or interruption. This prevents sensitive data from being accidentally exposed through leftover or cached files.

The application also includes error handling routines to detect invalid passwords, unsupported file types, or corrupted files. This contributes to the overall robustness and reliability of the tool.

IV. RESULTS

The developed encryption system was evaluated through a series of test cases designed to assess its functionality, performance, and usability. Tests focused on encryption and decryption accuracy, password validation, processing time, and compatibility with different file types. The experiments were conducted on a personal computing system under typical user conditions.

A. Experimental Setup

The evaluation of the proposed encryption system was conducted on a standard personal computer equipped with an Intel Core i5 processor clocked at 2.5 GHz, 8 GB of RAM, and running Windows 11. The application was implemented in Python 3.11 and executed as a standalone script without the need for administrative privileges or external dependencies beyond standard Python libraries. Testing focused on several key metrics to assess system performance and reliability. These included the time taken to encrypt and decrypt files of varying sizes and types, the accuracy of restoring original

file content following decryption, the robustness of the application when encountering invalid or tampered inputs, and the overall usability of the graphical user interface from an end-user perspective. Through this comprehensive setup, the system's functionality and efficiency were examined under typical operating conditions.

B. Performance Analysis

The system showed consistent performance across different file types and sizes. Encryption and decryption were successful in all valid test cases. The table below summarizes average processing times:

TABLE I
ENCRYPTION AND DECRYPTION PERFORMANCE

File Type	Size	Encrypt time	Decrypt Time (s)
Text (.txt)	3 KB	0.02	0.01
Image (.jpg)	1.1 MB	0.38	0.36
Document (.pdf)	2.5 MB	0.63	0.61
Video (.mp4)	15 MB	1.84	1.78

The processing time increased linearly with file size, as expected for symmetric encryption algorithms. For typical file sizes (under 5 MB), the application completed operations in under one second.

C. Functional Accuracy

The encryption and decryption procedures were verified by comparing checksums of the original and decrypted files. In all test cases, the decrypted files matched the originals byte-for-byte, indicating correct restoration of content and absence of data corruption.

The application also successfully rejected decryption attempts using incorrect passwords, thereby confirming that password-derived keys are securely and reliably implemented.

D. Error Handling and Robustness

To evaluate the robustness of the system, several negative test cases were executed to simulate potential user errors and tampering attempts. These included attempts to decrypt files that were never encrypted by the application, files that had been altered after encryption such as through modification of their initialization vectors or content, and the use of incorrect or empty passwords during decryption. In all cases, the application successfully identified inconsistencies and responded appropriately by displaying clear error messages, thereby preventing unauthorized access and system crashes. This behavior confirms the system's ability to gracefully handle unexpected or malicious input, ensuring that the underlying cryptographic operations remain secure even under abnormal usage scenarios.

V. CONCLUSION AND FUTURE WORK

This paper presented a secure and user-friendly file encryption system developed using Python, employing AES-256. in combination with password-based authentication to ensure data confidentiality. The application demonstrated that strong cryptographic principles can be effectively combined with an intuitive interface to deliver reliable protection for a wide range of file types. By deriving encryption keys from user-provided passwords through SHA-256 hashing, the system eliminates the need for key storage or management, thereby reducing the attack surface and simplifying the user experience. The inclusion of a lightweight GUI built with Tkinter further enhances accessibility, making the tool suitable for users with limited technical expertise.

Experimental results confirmed the system's efficiency, with encryption and decryption operations executing quickly even for larger files. Decryption using incorrect or corrupted input was reliably detected and rejected, demonstrating robustness against invalid operations and potential tampering. Furthermore, user feedback highlighted the practicality and simplicity of the interface, along with suggestions for further improving usability.

Looking ahead, several avenues exist for enhancing the system. These include adding support for drag-and-drop file operations, integrating progress indicators for long-running tasks, and enabling optional overwriting of original files during encryption. From a security perspective, incorporating advanced key derivation techniques such as PBKDF2 or Argon2 could further improve resistance against brute-force attacks. The system may also benefit from the inclusion of file integrity verification using HMAC, as well as expanded features for batch file processing or cloud synchronization. With these enhancements, the application has the potential to serve as a reliable tool not only for personal data protection but also in educational or organizational environments where secure, offline encryption is essential.

REFERENCES

[1] N. Kumar, S. Ghuge, and C. D. Jaidhar, "Three-layer security for password protection using rdh, aes and ecc," in *Hybrid Intelligent Systems (HIS)*. Cham: Springer, 2020, pp. 245–256.
[2] P. Banga *et al.*, "A secure file encryption and decryption system using aes for text and images," *International Journal of Research - Granthaalayah*, vol. 11, no. 12, pp. 254–267, Dec 2023.
[3] A. Aji and P. Pangestu, "Secure file sharing using advanced encryption standard (aes) 256," IT Telkom Purwokerto, Technical Report, Mar 2022.
[4] R. Ganesh, K. Niraj, and B. K. Das, "A novel framework for secure file transmission using modified aes and md5 algorithms," *International Journal of Information and Computer Security*, 2015.
[5] N. W. Nafi *et al.*, "A newer user authentication, file encryption and distributed server based cloud computing security architecture," *arXiv*, Mar 2013.
[6] S. Zainudin *et al.*, "Securing academic student file using aes algorithm for cloud storage web-based system," in *Reimagining Resilient Sustainability, European Proceedings of Multidisciplinary Sciences*, 2022, pp. 268–279.
[7] Wikipedia, "Advanced encryption standard," https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, 2024, online; updated recently.
[8] ——, "Pbkdf2," https://en.wikipedia.org/wiki/PBKDF2, 2024, online; updated last month.
[9] ——, "scrypt," https://en.wikipedia.org/wiki/Scrypt, 2009, online; published Mar. 2009.
[10] ——, "Hkdf," https://en.wikipedia.org/wiki/HKDF, 2024, online; updated four months ago.
[11] ResearchGate, "Password-based encryption approach for securing sensitive data," https://www.researchgate.net/publication/Password_based_encryption_for_sensitive_data, 2020.

[12] Unknown, "Security enhanced image encryption using password-based aes algorithm," *International Journal of Engineering Research and Technology (IJERT)*, n.d., year unspecified.
[13] ——, "Secure file operations: Using advanced encryption standard for strong data protection," *International Journal of Software Science & Engineering (IIETA)*, 2023, approximate year.
[14] C. StackExchange, "How secure is it to use password as aes key?" https://crypto.stackexchange.com/questions/22861, 2015.
[15] N. S. Ayesha *et al.*, "Dynamic encryption-based cloud security model using facial image and password-based key generation for multimedia data," *arXiv*, May 2025.