PROJECT REPORT

File Sharing Using AES Encryption and try to crack it

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the

degree of **Bachelor of Technology**

In

Computer Science and Engineering

School of Engineering and Sciences

Submitted by

Harsha Kamakshigari AP22110010319

Under the Guidance of

Mr.Bhaskara Santhosh Egala

Assistant professor, Dept.of CSE

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

April, 2025

Project Title- File Sharing Using AES Encryption and try to crack it

1. Introduction

In an era where information is constantly exchanged over digital platforms, ensuring the confidentiality and integrity of files is of utmost importance. This project aims to build a secure file-sharing system using AES (Advanced Encryption Standard), one of the most widely used symmetric encryption algorithms. Additionally, the project includes an educational attempt to crack the encrypted files by introducing known vulnerabilities such as static keys and weak initialization vectors (IVs). The purpose is to understand the implementation of secure encryption practices and identify what can go wrong if best practices are not followed.

2. Objectives

- To develop a web-based file-sharing application that encrypts files using AES before storage or transmission.
- To implement AES encryption and decryption in a way that is educational and demonstrative.
- To intentionally include and later exploit weak cryptographic configurations to understand vulnerabilities.
- To provide an intuitive user interface for uploading and downloading files securely.
- To demonstrate containerization and deployment practices using Docker and GitHub Actions.

3. Technologies Used

The project was developed using Python as the core language. FastAPI was used to build the web backend, while HTML and Tailwind CSS were used to develop a simple and responsive frontend. The `cryptography` Python library provided support for AES encryption. Docker was used to containerize the application, and GitHub Actions was integrated to enable a basic CI/CD pipeline.

4. System Design and Implementation

AES Encryption Logic

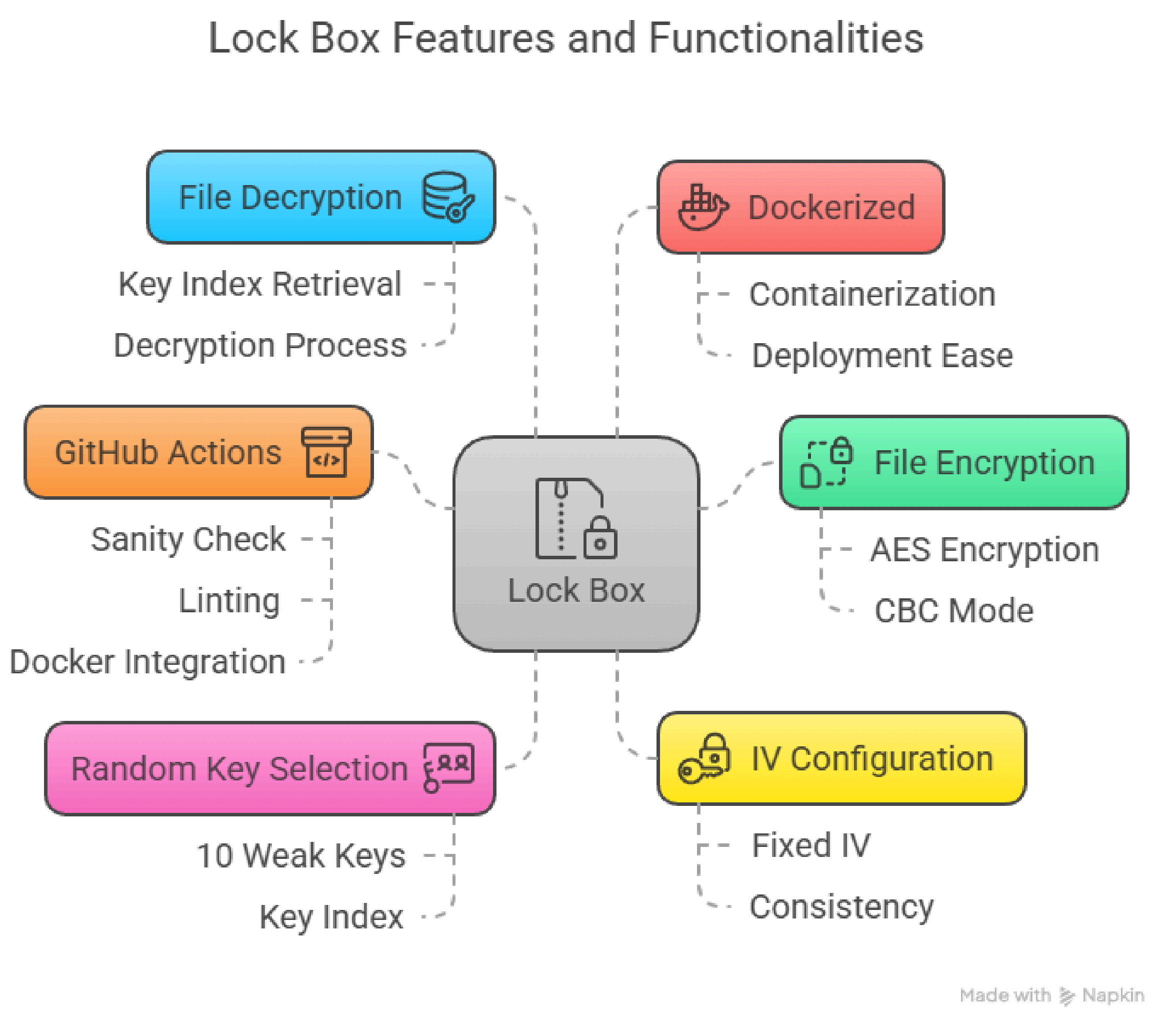
The application uses AES in Cipher Block Chaining (CBC) mode for encrypting files. A symmetric key and initialization vector (IV) are required for both encryption and decryption. For educational purposes, the system uses a hardcoded key and a known weak IV. This design helps demonstrate the risks involved when cryptographic best practices are not followed.

Steps:

- A file is uploaded through the web interface.
- The file data is read in binary format.
- AES encryption is applied using a static 256-bit key and IV.
- The encrypted file is saved to a local server directory.
- Metadata is stored for referencing the file during decryption.

Decryption and File Download

- The user can download an encrypted file through the interface.
- Upon request, the file is decrypted using the same static key and IV.
- The original file is reconstructed and made available for download.



5. Cracking Attempt

To highlight the importance of secure key and IV management, a cracking simulation was implemented. This involved:

- Attempting a brute-force approach using a small dictionary of possible keys.
- Attempting to reverse the encryption by guessing the IV (since it was weak and known).
- Using file signature analysis to identify decrypted file formats (like PDF, PNG, TXT).

These attacks showed that if a system uses predictable keys or IVs, encryption can be bypassed even if a strong algorithm like AES is used.

5.1 How a Hacker Might Attempt to Decrypt the File

In the absence of secure key management, an attacker can try to decrypt AES-encrypted files using a combination of strategic techniques and powerful tools. Here's a breakdown of how such an attack might proceed in practice:

Step-by-Step Approach of an Attacker

1. File Inspection and Metadata Analysis

Goal: Understand file structure, size, and possible formats.

- **Tools used:**
  - `xxd` – to view the hexadecimal dump of the encrypted file.
  - `file` (Linux utility) – to detect file type, even if partially encrypted.
  - `binwalk` – for identifying embedded file signatures within encrypted blobs.
  - `hexdump` or `HxD` (on Windows) – for binary analysis of patterns.

Use Case Example:
The attacker observes the beginning bytes of the file to check for signatures like `%PDF`, `PNG`, or `PK` (ZIP), helping them identify correct decryption guesses.

2. Brute-Force or Dictionary Attack on Static Key

Goal: Guess the key used for encryption.

- **Tools used:**
  - **Hashcat** – a high-performance tool for brute-forcing or dictionary attacks.
  - **John the Ripper** – another password-cracking tool, which can be extended with plugins for symmetric encryption.
  - **Custom Python Scripts** using `PyCryptodome` or `cryptography` for testing keys in loops.
  - **Hydra (if password-based encryption)** – for automating login or password brute-force on systems.

Use Case Example:
If the attacker knows the AES key is 256-bit but weak or hardcoded, they write a script to loop through a small wordlist of likely candidates and decrypt the file until they see a valid output file.

3. Guessing or Extracting the Initialization Vector (IV)

Goal: Retrieve or guess the IV used in CBC mode.

- **Scenario:**
  - If the IV is static or fixed, the attacker can hardcode it into their script.
  - If it's included in the encrypted file (e.g., first 16 bytes), they extract it directly.
- **Tools used:**
  - **Wireshark (in transmission attacks)** – to sniff and extract IVs from traffic.
  - **Custom Scripts** – to parse or guess IVs using known patterns.

Use Case Example:
An attacker assumes the IV is either `16 zero bytes` or `all '1's` and tests those values during decryption.

4. Plaintext Pattern Matching

Goal: Detect if the output of decryption is a valid file.

- **Tools used:**
  - **foremost** or **scalpel** – for carving known file types from partial decrypted data.
  - **Magic number databases** – to identify the beginning of known file formats.
  - **Strings** command – to extract readable ASCII data from binary files.

Use Case Example:
After each decryption attempt, the attacker runs `strings` on the output to check if common terms appear (e.g., "PDF", "Title", "Name").

5. Automation of Attack with Scripts

Goal: Automate multiple decryption attempts with changing keys or IVs.

- **Tools used:**
  - **Bash/Python scripting** – to loop over a set of keys or IVs.

    - **OpenSSL CLI** – for manual or scripted AES decryption attempts.
    - **Aircrack-ng/Crunch** – to generate custom wordlists or keys.

6. Frontend and User Interface

A lightweight frontend was created using HTML and Tailwind CSS. Users can upload and download files through simple forms. No login or user authentication was included as the focus was primarily on encryption and security logic. However, this could be expanded in future versions.

7. Dockerization and CI/CD Integration

The application was containerized using Docker to ensure consistency across development and deployment environments. GitHub Actions was configured to automate testing and build steps, allowing changes to be automatically deployed after each push to the main branch.

8. Challenges Faced

- Managing large file sizes during encryption and maintaining performance.
- Implementing encryption in a secure way while also allowing for demonstration of weaknesses.
- Creating an intuitive frontend that aligns well with the backend logic.
- Simulating attacks in a controlled and safe environment without using advanced cracking tools.

9. Learning Outcomes

- Gained hands-on experience with AES encryption and the importance of secure key management.
- Understood how encryption can be broken when best practices are ignored.

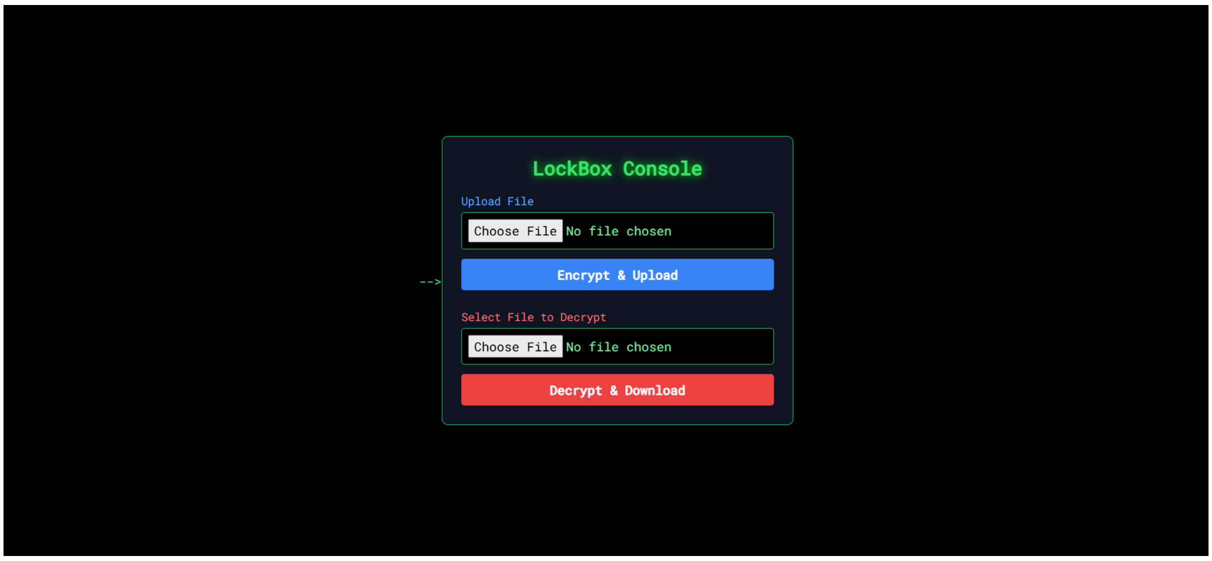
- Learned how to build and deploy a complete application using FastAPI, Docker, and GitHub Actions.
- Improved understanding of real-world file security and cryptographic principles.

10. Future Enhancements

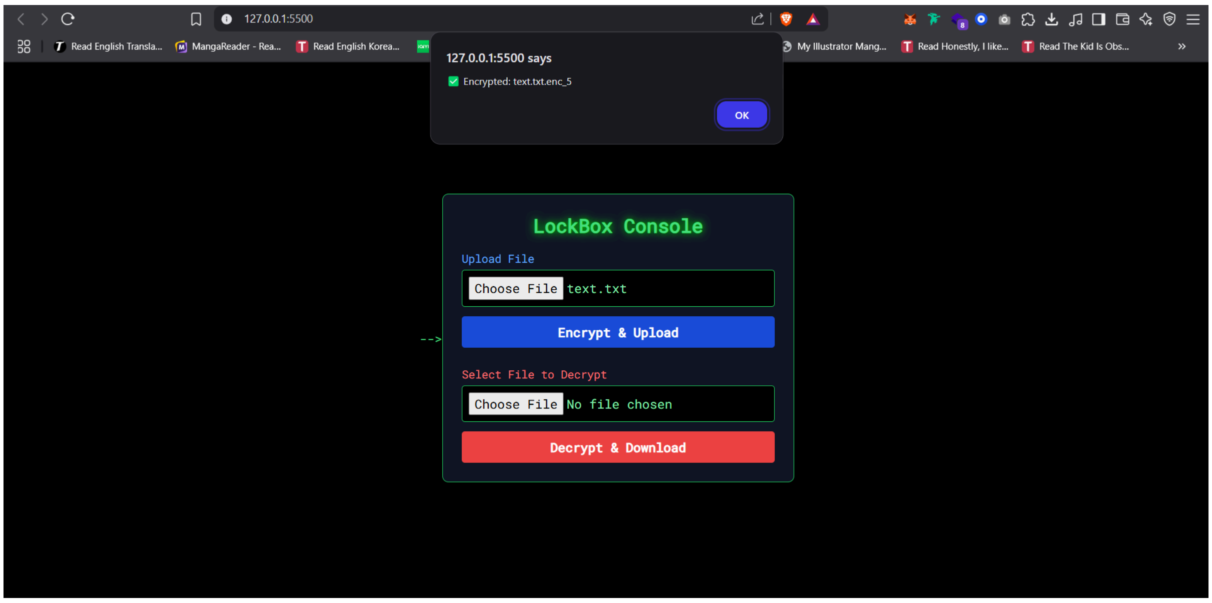
- Implementing user authentication and access control for secure file ownership.
- Adding expiration policies for file sharing links.
- Using dynamic, randomly generated keys and IVs stored securely.
- Integrating cloud storage support for scalability.
- Providing real-time encryption status and progress feedback to users.

11.Evidence and Screenshots

Interface



Encryption



127.0.0.1:5500 says

Encrypted: text.txt.enc_5

LockBox Console

Upload File

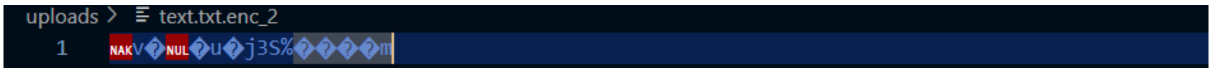
Choose File text.txt

Encrypt & Upload

Select File to Decrypt

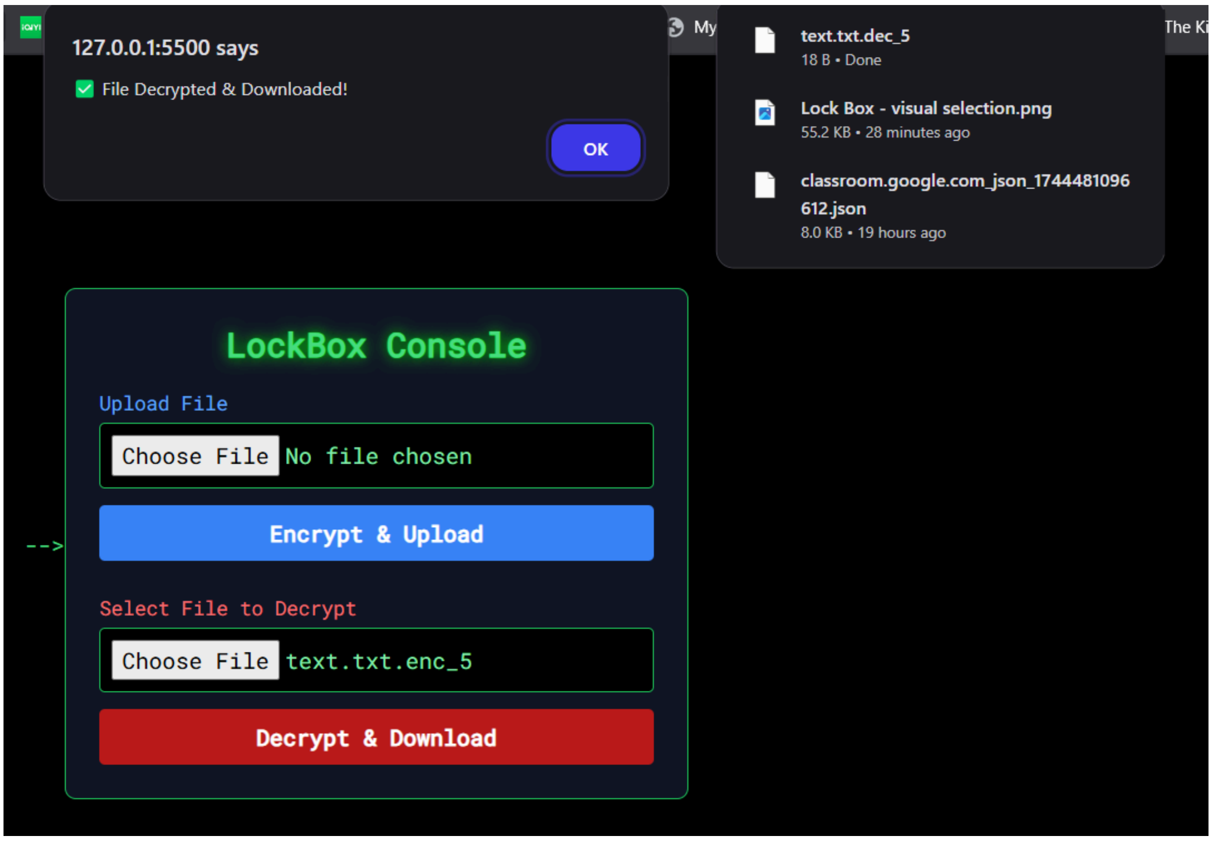
Choose File No file chosen

Decrypt & Download



```
uploads > text.txt.enc_2
1   NAKV�NUL�u�j3S%����m
```

Decryption



127.0.0.1:5500 says

File Decrypted & Downloaded!

text.txt.dec_5
18 B • Done

Lock Box - visual selection.png
55.2 KB • 28 minutes ago

classroom.google.com_json_1744481096612.json
8.0 KB • 19 hours ago

LockBox Console

Upload File

Choose File No file chosen

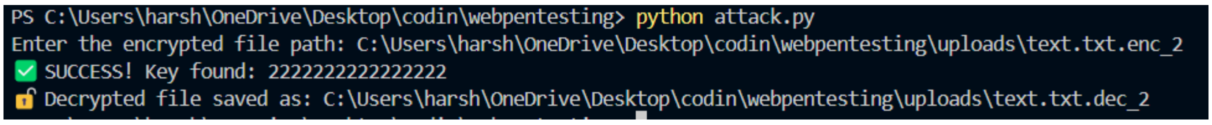
Encrypt & Upload

Select File to Decrypt

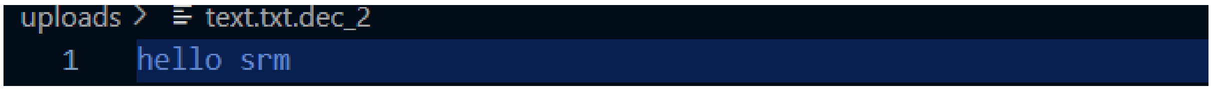
Choose File text.txt.enc_5

Decrypt & Download

Dictionary Attack



```
PS C:\Users\harsh\OneDrive\Desktop\codin\webpentesting> python attack.py
Enter the encrypted file path: C:\Users\harsh\OneDrive\Desktop\codin\webpentesting\uploads\text.txt.enc_2
✅ SUCCESS! Key found: 2222222222222222
🔓 Decrypted file saved as: C:\Users\harsh\OneDrive\Desktop\codin\webpentesting\uploads\text.txt.dec_2
```



```
hello srm
```

12. Conclusion

This project successfully demonstrated the creation of a secure file-sharing application using AES encryption. By also exploring how weak implementations can be exploited, it provided a comprehensive understanding of both the strengths and vulnerabilities of symmetric cryptography. The application highlights the balance between ease of use and security, emphasizing the importance of following cryptographic best practices in real-world applications.

Demo video link