SECURE FILE SHARING SYSTEM

Submitted by: Hari Haran

Date: 17/10/2025

Project Overview

The Secure File Sharing System is a backend-based application that allows users to securely share files by encrypting them with a strong AES (Advanced Encryption Standard) algorithm. The project ensures data confidentiality and integrity when files are stored or transferred. It uses AES-GCM mode with password-based key derivation to ensure that only authorized users can decrypt shared files.

Objectives

• Design a secure backend system for file encryption and decryption.

• Ensure data confidentiality, authenticity, and integrity using AES encryption.

• Protect files both at rest and in transit.

• Demonstrate the use of modern cryptography in real-world applications.

Technologies Used

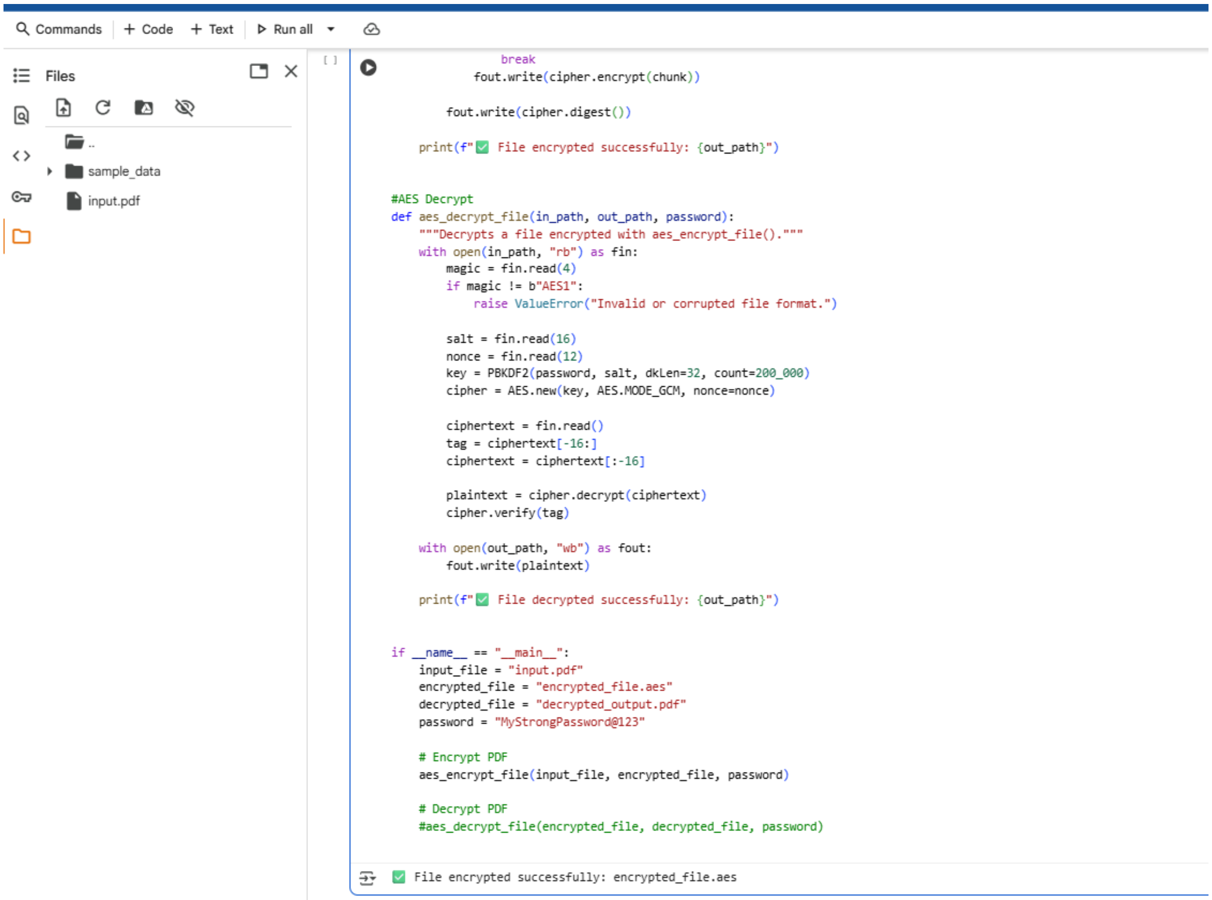
Component	Description
Language	Python
Libraries	PyCryptodome (for AES and KDF)
Algorithm	AES-GCM (Advanced Encryption Standard – Galois Counter Mode)
Key Derivation	PBKDF2 with salt
Random Generator	get_random_bytes() for salt & nonce
IDE/Platform	VS Code / PythonAnywhere / Local System

Working Principle

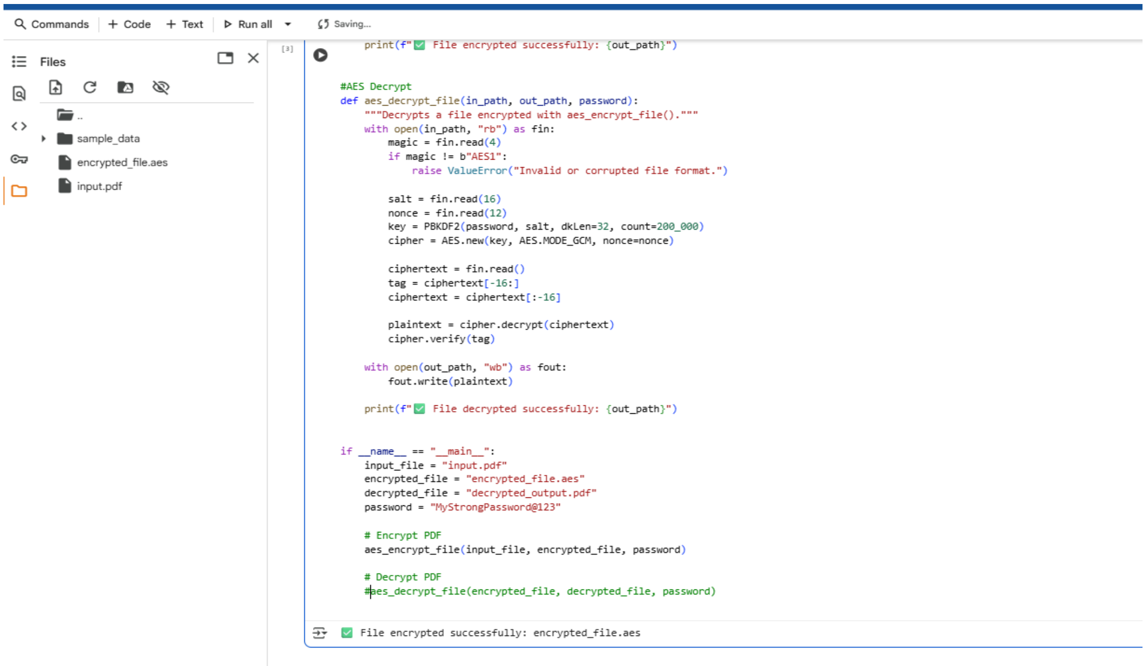
The system performs encryption and decryption using AES-GCM mode as follows:

1. Encryption Process:

- User provides a file and password.
- Random salt and nonce are generated.
- A 256-bit AES key is derived using PBKDF2.
- File is encrypted in chunks and stored with authentication tag.




```python
                break
            fout.write(cipher.encrypt(chunk))

        fout.write(cipher.digest())

    print(f"✅ File encrypted successfully: {out_path}")


#AES Decrypt
def aes_decrypt_file(in_path, out_path, password):
    """Decrypts a file encrypted with aes_encrypt_file()."""
    with open(in_path, "rb") as fin:
        magic = fin.read(4)
        if magic != b"AES1":
            raise ValueError("Invalid or corrupted file format.")

        salt = fin.read(16)
        nonce = fin.read(12)
        key = PBKDF2(password, salt, dkLen=32, count=200_000)
        cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)

        ciphertext = fin.read()
        tag = ciphertext[-16:]
        ciphertext = ciphertext[:-16]

        plaintext = cipher.decrypt(ciphertext)
        cipher.verify(tag)

    with open(out_path, "wb") as fout:
        fout.write(plaintext)

    print(f"✅ File decrypted successfully: {out_path}")


if __name__ == "__main__":
    input_file = "input.pdf"
    encrypted_file = "encrypted_file.aes"
    decrypted_file = "decrypted_output.pdf"
    password = "MyStrongPassword@123"

    # Encrypt PDF
    aes_encrypt_file(input_file, encrypted_file, password)

    # Decrypt PDF
    #aes_decrypt_file(encrypted_file, decrypted_file, password)
```

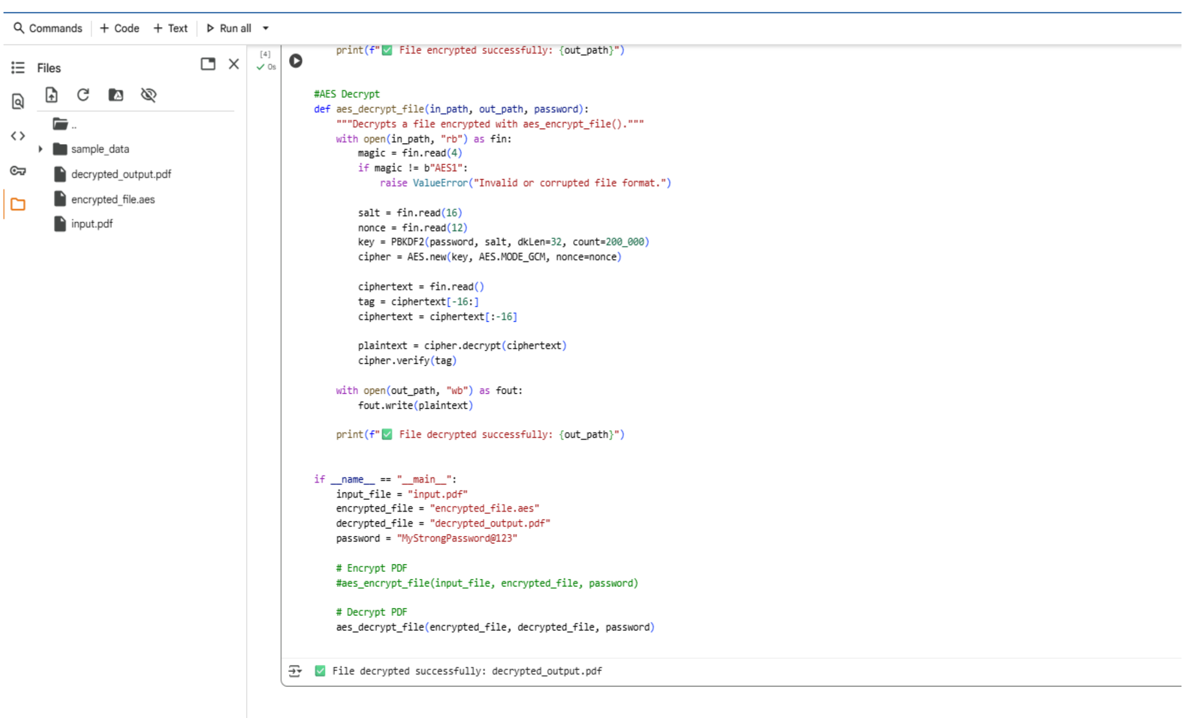


```python
    print(f"✅ File encrypted successfully: {out_path}")

#AES Decrypt
def aes_decrypt_file(in_path, out_path, password):
    """Decrypts a file encrypted with aes_encrypt_file()."""
    with open(in_path, "rb") as fin:
        magic = fin.read(4)
        if magic != b"AES1":
            raise ValueError("Invalid or corrupted file format.")

        salt = fin.read(16)
        nonce = fin.read(12)
        key = PBKDF2(password, salt, dkLen=32, count=200_000)
        cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)

        ciphertext = fin.read()
        tag = ciphertext[-16:]
        ciphertext = ciphertext[:-16]

        plaintext = cipher.decrypt(ciphertext)
        cipher.verify(tag)

    with open(out_path, "wb") as fout:
        fout.write(plaintext)

    print(f"✅ File decrypted successfully: {out_path}")


if __name__ == "__main__":
    input_file = "input.pdf"
    encrypted_file = "encrypted_file.aes"
    decrypted_file = "decrypted_output.pdf"
    password = "MyStrongPassword@123"

    # Encrypt PDF
    aes_encrypt_file(input_file, encrypted_file, password)

    # Decrypt PDF
    #aes_decrypt_file(encrypted_file, decrypted_file, password)
```

```
✅ File encrypted successfully: encrypted_file.aes
```


2. Decryption Process:

- User provides the encrypted file and password.
- Salt and nonce are read to regenerate the AES key.
- File is decrypted and verified using authentication tag.
- Original file is restored successfully.



```python
    print(f"✅ File encrypted successfully: {out_path}")

#AES Decrypt
def aes_decrypt_file(in_path, out_path, password):
    """Decrypts a file encrypted with aes_encrypt_file()."""
    with open(in_path, "rb") as fin:
        magic = fin.read(4)
        if magic != b"AES1":
            raise ValueError("Invalid or corrupted file format.")

        salt = fin.read(16)
        nonce = fin.read(12)
        key = PBKDF2(password, salt, dkLen=32, count=200_000)
        cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)

        ciphertext = fin.read()
        tag = ciphertext[-16:]
        ciphertext = ciphertext[:-16]

        plaintext = cipher.decrypt(ciphertext)
        cipher.verify(tag)

    with open(out_path, "wb") as fout:
        fout.write(plaintext)

    print(f"✅ File decrypted successfully: {out_path}")


if __name__ == "__main__":
    input_file = "input.pdf"
    encrypted_file = "encrypted_file.aes"
    decrypted_file = "decrypted_output.pdf"
    password = "MyStrongPassword@123"

    # Encrypt PDF
    #aes_encrypt_file(input_file, encrypted_file, password)

    # Decrypt PDF
    aes_decrypt_file(encrypted_file, decrypted_file, password)
```

```
✅ File decrypted successfully: decrypted_output.pdf
```

SECURE FILE SHARING SYSTEM

Code Summary

The Python script imports AES, PBKDF2, and random byte generation modules from the PyCryptodome library. The aes_encrypt_file() function encrypts files securely using AES-GCM, while aes_decrypt_file() restores the original file after password verification. This ensures that data confidentiality and integrity are maintained.

Example Execution

Input File: input.pdf
Password: MyStrongPassword@123

Output Files:
• Encrypted: encrypted_file.aes
• Decrypted: decrypted_output.pdf

Result: File encrypted and decrypted successfully — contents match the original file.

Security Features

• Uses AES-GCM to ensure encryption and message integrity.

• Salt + PBKDF2 prevent brute-force or rainbow table attacks.

• Unique nonce ensures different encryption each time.

• Authentication tag detects unauthorized modification.

Advantages

• Strong cryptographic protection.

• Lightweight backend — no frontend required.

• Easy to integrate with web APIs or cloud systems.

• Protects sensitive files effectively.

Future Enhancements

• Add a web-based frontend for user uploads and downloads.

• Include user authentication for multi-user access.

• Integrate with cloud-based encrypted file storage.

• Add automatic file deletion after successful sharing.

Conclusion

This project successfully demonstrates secure file encryption and decryption using AES-GCM and PBKDF2. It highlights the importance of cryptography in protecting sensitive data during sharing and storage. The Secure File Sharing System ensures confidentiality, authenticity, and integrity — serving as a strong foundation for building secure data-sharing applications.