# **Text Encryption Using Various Algorithms**

**Manvi Sethi - 500120341**



Submitted in Partial fulfillment of the Degree of Bachelor of Technology (B. Tech) in Computer Science Engineering (H)

**UNIVERSITY OF PETROLEUM AND ENERGY STUDIES**

**Contents**

[**Text Encryption Using Various Algorithms** 1](#_Toc204608367)

[DECLARATION 3](#_Toc204608368)

[ACKNOWLEDGEMENT 4](#_Toc204608369)

[Abstract: Secure File Encryption and Key Transmission System Using AES and Flask 5](#_Toc204608370)

[Purpose of the Project 7](#_Toc204608371)

[ALGORITHM 1: ADVANCED ENCRYPTION STANDARD 8](#_Toc204608372)

[Introduction to AES (Advanced Encryption Standard) 8](#_Toc204608373)

[Libraries Used 10](#_Toc204608374)

[Secure File Encryption and Key Exchange Using AES and Flask 13](#_Toc204608375)

[Abstract 13](#_Toc204608376)

[Key Components: 14](#_Toc204608377)

[Objective and Goals of AES 16](#_Toc204608378)

[**Methodology (Detailed)** 18](#_Toc204608379)

[Vulnerabilities, Threats and Backdoors 22](#_Toc204608380)

[1. Vulnerabilities in the Current System 22](#_Toc204608381)

[2. Threats 22](#_Toc204608382)

[3. Backdoors 23](#_Toc204608383)

[Mitigation Techniques 24](#_Toc204608384)

[Overall Description Of The Project 26](#_Toc204608385)

[Implementation 29](#_Toc204608386)

[Screenshots 31](#_Toc204608387)

[ALGORITHM 2: CIPHER CAESER 33](#_Toc204608388)

[Vulnerabilities, Threats, and Backdoors in Caesar Cipher 35](#_Toc204608389)

[Mitigation Techniques 37](#_Toc204608390)

[Implementation 39](#_Toc204608391)

[Screenshots 42](#_Toc204608392)

[**Problem Statement** 43](#_Toc204608393)

[**Limitations** 44](#_Toc204608394)

[CONCLUSION 4](#_Toc204608395)7

[References 4](#_Toc204608396)8

**DECLARATION**

I , Manvi Sethi, hereby declare that this submission is my own work and that to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

**Date: 23 July 2025 MANVI SETHI**

**Place: Delhi, India**

**ACKNOWLEDGEMENT**

I wish to express my deep gratitude to Mr. Mohit Jain, for all the advice, encouragement and constant support he has given me throughout the project work.

This work would not have been possible without his support and valuable suggestions. I am extremely grateful to the supervisor, Mr. Kashif Ahmed, for his invaluable guidance, encouragement and support during my time here. His insights and advice have pushed me to grow and learn in ways I didn't think possible. I could not have asked for a better mentor.

I am grateful to the MS IT infra team for giving me the necessary facilities to carry out the project work successfully and for providing timely support and information during the completion of the project. I would also like to thank the wonderful team at Power Finance Corporation.

Thank you for making me feel so welcomed and for patiently answering all my questions. Your passion and dedication to your work is truly inspiring. This internship has been a defining period in my life and I could not have done it without all of your help.

Thank you,

**Manvi Sethi**

# Abstract: Secure File Encryption and Key Transmission System Using AES and Flask

In the modern digital era, securing sensitive data during storage and transmission is a critical challenge. This project presents the design and implementation of a **secure file encryption and key exchange system** using **AES (Advanced Encryption Standard)** in **CBC (Cipher Block Chaining)** mode, integrated with a **Flask-based REST API** for secure key transmission.

The system performs **file encryption** on the client side using **AES-128**, with a **randomly generated 128-bit symmetric key** and **16-byte Initialization Vector (IV)** created through Python’s cryptographically secure random number generator. These values are encoded in **Base64** to ensure safe transport over HTTP and are transmitted via an **HTTPS POST request** to a Flask server endpoint (/upload-key). The server, running with **SSL/TLS encryption**, receives, decodes, and logs the key and IV for later decryption processes.

The project is modularly developed with three main components:

1. **AES Encryption/Decryption Module** – Implements file encryption using AES with PKCS7-style padding and CBC mode; allows secure decryption using the same key and IV.
2. **Key Generation and Sender Script** – Generates cryptographic keys, encodes them with Base64, and securely transmits them using the requests library.
3. **Flask-based Key Receiver API** – Implements a secure API server to receive keys over HTTPS, simulating a real-world key exchange mechanism.

In addition, the project also explores the **Caesar Cipher**, a classical encryption technique, to compare traditional and modern encryption standards. It includes implementation of Caesar Cipher with additional support for digits and symbols, as well as a vulnerability analysis and mitigation suggestions.

Security best practices such as **SSL usage**, **key confidentiality**, **input validation**, and **rate limiting** are incorporated into the design. The project also identifies potential **vulnerabilities and threats** like man-in-the-middle attacks, lack of API authentication, and insecure key management, and proposes effective **mitigation techniques** including HTTPS enforcement, proper key storage, and authentication mechanisms.

This project offers practical insight into the implementation of secure cryptographic systems using modern tools and languages. It serves as a foundational model for secure file transmission applications, with potential for future enhancements like file uploads, user authentication, and cloud storage integration.

**Purpose of the Project**

The purpose of this project is to develop a secure and reliable data communication system that enables encrypted exchange of information between a client and a server using AES (Advanced Encryption Standard) encryption and a Flask-based API. The system is designed to ensure that sensitive data such as files, messages, or cryptographic keys are transmitted securely over the internet without the risk of unauthorized access or tampering. By integrating AES encryption for confidentiality and Flask for server-side communication, the project aims to provide a simple yet powerful framework that guarantees data privacy, integrity, and safe transmission.

This system can be used in various scenarios where secure communication is essential, such as file transfers, encrypted messaging, or protected data storage over networks.

# **ALGORITHM 1: ADVANCED ENCRYPTION STANDARD**

## **Introduction to AES (Advanced Encryption Standard)**

**AES (Advanced Encryption Standard)** is a widely used **symmetric encryption algorithm** that is used to **secure data** in various applications such as communication, storage, and networking. It was adopted as a standard by the **U.S. National Institute of Standards and Technology (NIST)** in **2001** to replace the older **Data Encryption Standard (DES)** due to its weaknesses.

**Key Features of AES:**

1. **Symmetric Key Encryption**:
   * The same key is used for both **encryption** and **decryption**.
2. **Block Cipher**:
   * AES operates on **blocks of data** — specifically, **128-bit blocks** (16 bytes).
3. **Key Sizes**:
   * Supports three key lengths:
     + **AES-128**: 128-bit key (10 rounds)
     + **AES-192**: 192-bit key (12 rounds)
     + **AES-256**: 256-bit key (14 rounds)
4. **Rounds**:
   * A "round" consists of several steps: **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKey**.

**How AES Works: (Simplified Steps)**

1. **Key Expansion**:
   * The original key is expanded into multiple round keys.
2. **Initial Round**:
   * AddRoundKey (XOR the data with a round key)
3. **Main Rounds (repeated)**:
   * **SubBytes**: Substitute each byte using an S-box
   * **ShiftRows**: Rotate rows of the matrix
   * **MixColumns**: Mix bytes in each column
   * **AddRoundKey**: XOR with the round key
4. **Final Round**:
   * SubBytes, ShiftRows, AddRoundKey (no MixColumns)

**Why AES is Secure:**

* Uses **strong mathematical operations** (like substitution and permutation).
* Resistant to common attacks (e.g., brute force, linear cryptanalysis).
* Efficient in both **hardware and software**.

# **Libraries Used**

**1. from Crypto.Cipher import AES**

**Library: pycryptodome**

**Purpose: Provides AES encryption and decryption functionality.**

* AES.new(key, AES.MODE\_CBC, iv)  
  ➤ Creates a new AES cipher object using the key and IV in **CBC (Cipher Block Chaining)** mode.
* cipher.encrypt(pad(data))  
  ➤ Encrypts the data (after padding) using the AES cipher.
* cipher.decrypt(encrypted)  
  ➤ Decrypts the encrypted data.

**2. from Crypto.Random import get\_random\_bytes**

**Library: pycryptodome**

**Purpose: Generates cryptographically secure random bytes.**

* get\_random\_bytes(16)  
  ➤ Generates a 16-byte random key or IV. Secure for cryptographic use.

**3. import base64**

**Library: Python Standard Library**

**Purpose: Converts binary data (key/IV) to base64 string and vice-versa.**

* base64.b64encode(key).decode()  
  ➤ Converts byte data into base64-encoded string (useful for display or sending via network).
* base64.b64decode(key\_string)  
  ➤ Converts base64 string back to byte format (required for AES).

Base64 encoding is used because keys and IVs are in bytes format and not readable/transmittable easily as-is.

**4. from flask import Flask, request, jsonify**

**Library: Flask (web framework)**

**Purpose: Creates a web API endpoint to receive the key and IV.**

* Flask(\_\_name\_\_)  
  ➤ Initializes a Flask web app.
* @app.route('/upload-key', methods=['POST'])  
  ➤ Defines an API endpoint that accepts POST requests at /upload-key.
* request.json  
  ➤ Parses the incoming JSON payload.
* jsonify({...})  
  ➤ Sends a JSON response back to the client.

Flask is used here to simulate a **simple secure server** that receives AES encryption keys from a client.

**5. Client-Side: Key Sender Script**

**Libraries Used:**

| **Library** | **Purpose** |
| --- | --- |
| requests | To send HTTP requests to the Flask API (POST, GET, etc.) |
| base64 | To safely encode binary data (key and iv) into a text format |
| Crypto.Random | To generate a secure 16-byte key and IV for AES encryption |

# **Secure File Encryption and Key Exchange Using AES and Flask**

## Abstract

This project focuses on implementing a secure file encryption and decryption system using the AES (Advanced Encryption Standard) algorithm in Cipher Block Chaining (CBC) mode, integrated with a Flask-based web server for safe key exchange.

The encryption process uses a randomly generated 128-bit key and a 16-byte Initialization Vector (IV), both created using Python’s cryptographic random generator. These values are Base64 encoded for safe transmission and sent to the server via an HTTPS POST request using the requests library. The server, developed using Flask, receives the key and IV through a /upload-key endpoint and logs them securely. The system also supports decryption using the same key and IV to retrieve the original file.

This project combines key concepts of symmetric encryption, secure communication, and REST API development to build a foundational model for secure data handling and transmission in Python.

## **Key Components**

**1. AES Encryption and Decryption Module**

* **Technology Used:** pycryptodome (Crypto.Cipher)
* **Functionality:**
  + Reads file content in binary mode.
  + Applies **PKCS-style padding** to align data into 16-byte blocks.
  + Uses **AES-128 in CBC mode** to encrypt data.
  + Saves encrypted file with .enc extension.
  + Also supports decryption by reversing the process using the same key and IV.

**2. Key and IV Generator + Sender (Client Script)**

* **Technology Used:** Crypto.Random, base64, requests
* **Functionality:**
  + Generates a 128-bit (16-byte) random key and a 16-byte Initialization Vector (IV) using get\_random\_bytes().
  + Encodes both key and IV using **Base64** so they can be safely transferred as text.
  + Sends key and IV to the server securely via **HTTPS POST request** using the requests library.
  + For testing, SSL certificate verification can be disabled (verify=False), but must be enabled in production (verify=True).

**3. Flask-Based Secure Key Receiver (API Server)**

* **Technology Used:** Flask (Python Web Framework)
* **Functionality:**
  + Hosts a REST API endpoint /upload-key which listens for POST requests.
  + Receives JSON data containing Base64-encoded AES key and IV.
  + Decodes and logs them securely (for development/testing purposes).
  + Runs over **HTTPS** using a self-signed certificate (cert.pem, key.pem).
  + Confirms key receipt using a JSON response like { "message": "Key received successfully" }.

## **Objective and Goals of AES**

**Objective**

The primary objective of this project is to develop a secure system for file encryption and decryption using symmetric key cryptography, specifically the AES (Advanced Encryption Standard) algorithm in CBC (Cipher Block Chaining) mode. The system also includes a secure mechanism for transmitting encryption keys and Initialization Vectors (IVs) from the client to a server using a Flask-based REST API over HTTPS. This project aims to ensure confidentiality of sensitive file data, secure key exchange, and demonstrate a real-world application of cryptography integrated with secure web communication.

**Goals**

To achieve the above objective, the following goals have been set:

1. **Implement AES-based encryption and decryption**
   * Use AES-128 in CBC mode to securely encrypt and decrypt file data.
   * Handle padding to ensure compatibility with block-based encryption.
2. **Generate secure cryptographic key and IV**
   * Use cryptographically secure random functions to generate a 128-bit key and a 16-byte IV.
3. **Transmit the key and IV securely over the network**
   * Encode key and IV using Base64 for safe transfer.
   * Send them using HTTPS to ensure confidentiality and prevent interception.
4. **Create a Flask API to receive encryption keys**
   * Design a /upload-key POST endpoint to accept key and IV from the client.
   * Run the Flask server with SSL/TLS support for secure communication.
5. **Demonstrate integration between cryptography and network communication**
   * Combine file encryption and RESTful APIs to build a practical, secure system.
   * Provide a working model that can be extended into real-world secure file sharing or data protection systems.
6. **Promote secure programming practices**
   * Highlight the importance of using encryption for data security.
   * Encourage the use of HTTPS and proper handling of sensitive data.

# **Methodology (Detailed)**

This project adopts a modular and phased development approach to implement secure file encryption using AES along with secure transmission of encryption keys through a Flask-based web API. Each phase of the project is described below:

**🔹 1. Technology Selection and Requirements Gathering**

* Identify the need to protect file data using encryption and safely share encryption keys.
* Decide on the use of **symmetric key encryption** for simplicity and performance.
* Choose **AES (Advanced Encryption Standard)** due to its widespread use and security strength.
* Choose **CBC (Cipher Block Chaining)** mode to provide randomness in encryption with an IV.
* Select **Flask** for creating a lightweight and easy-to-implement RESTful API server.
* Use **HTTPS** to ensure secure data transmission.

**🔹 2. File Encryption Module Development**

* **AES-128 in CBC Mode** is used for encryption.
* A function reads the file in **binary format** to handle any file type (text, image, etc.).
* Data is padded using a PKCS7-like scheme to make its length a multiple of 16 bytes.
* The padded data is encrypted using the AES.new(key, AES.MODE\_CBC, iv) method.
* The encrypted data is saved with a .enc extension using a new file write operation.

**🔹 3. File Decryption Module Development**

* Another function reads the encrypted .enc file.
* The AES cipher is initialized again with the **same key and IV**.
* The cipher decrypts the data and removes padding.
* The original file content is restored and saved with a new filename (e.g., filedecrypted.txt).

**🔹 4. Key and IV Generation**

* A 128-bit AES **encryption key** and a 16-byte **Initialization Vector (IV)** are generated using:
* These random values are **cryptographically secure**, ensuring the uniqueness of every encryption.
* The key and IV are encoded using **Base64 encoding** to make them suitable for transmission via HTTP in JSON format.

**🔹 5. Secure Key Transmission (Client-Side Script)**

* A client script is written using the requests library to send the Base64-encoded key and IV to the Flask server.
* The script sends a **POST request** to the API endpoint /upload-key with a JSON body containing:
* For testing, verify=False is used to bypass SSL certificate validation (not recommended in production).
* The server responds with a success message if the key and IV are received properly.

**🔹 6. Flask API Server Development**

* A Flask web server is created with a route /upload-key to handle POST requests.
* When the API receives a request:
  + It parses the JSON content using request.json.
  + Extracts the key and iv fields.
  + Decodes them from Base64.
  + Logs the values (in a real-world system, these should be securely stored).
* The server runs on **HTTPS** using self-signed certificates:

**🔹 7. SSL Certificate Setup**

* A self-signed certificate (cert.pem) and a private key (key.pem) are generated using OpenSSL.
* These certificates are used to serve the Flask API over **HTTPS**, ensuring encrypted transmission.
* This protects the key and IV from being intercepted over the network.

**🔹 8. Integration and Testing**

* All components (encryption, key transmission, and decryption) are tested end-to-end.
* The following steps are performed during testing:
  1. Encrypt a sample file.
  2. Generate and send the key and IV to the server.
  3. Use the same key and IV to decrypt the file and validate correctness.
  4. Test error handling: wrong key/IV, corrupted file, missing fields, etc.

**🔹 9. Security Considerations**

* The key and IV are never hardcoded — they are randomly generated every time.
* The communication between client and server is secured using **HTTPS**.
* In a production system:
  + SSL certificate validation should be enabled (verify=True).
  + Authentication and authorization mechanisms should be added.
  + Keys should be stored securely (e.g., in an encrypted database or a secrets vault).

**Conclusion of Methodology**

This methodology ensures a modular, secure, and scalable design that covers file encryption, secure key exchange, and network-level protection. It allows for easy future enhancements such as user authentication, file uploads, key storage, and more advanced encryption modes.

# 

# **Vulnerabilities, Threats and Backdoors**

## 1**. Vulnerabilities in the Current System**

A **vulnerability** is a weakness or flaw in a system that can be exploited by attackers.

| **Vulnerability** | **Explanation** |
| --- | --- |
| **No HTTPS validation (verify=False)** | In the client code, SSL verification is disabled using verify=False. This makes the system vulnerable to **MITM (Man-in-the-Middle) attacks**. |
| **Hardcoded or auto-generated keys without protection** | Keys and IVs are printed on screen and stored temporarily; if logs are exposed or terminal is accessible, keys can be stolen. |
| **No authentication for Flask API** | Anyone who sends a POST request to /upload-key can upload a key. There is no token, password, or user authentication. |
| **No file integrity check** | There is no hashing (e.g., SHA256) to verify if the encrypted/decrypted file is tampered or corrupted. |
| **Key transmission without encryption** | Although keys are Base64 encoded, **Base64 is not encryption**, so the key is still readable if intercepted. |
| **No file upload validation** | If extended to upload files in future, no checks exist to prevent malicious file uploads (like .exe, .bat, etc.). |

## 2. **Threats**

**Threats** are potential dangers that may exploit the vulnerabilities above.

| **Threat** | **Impact** |
| --- | --- |
| **Eavesdropping** | If someone is sniffing network traffic (especially on open Wi-Fi), and verify=False is used, they can intercept the key and IV. |
| **Unauthorized access to API** | Since there's no authentication, an attacker can spam the /upload-key endpoint or overwrite existing key logs. |
| **Key leakage** | If the attacker has access to the terminal or logs where the keys are printed, they can decrypt files later. |
| **Denial of Service (DoS)** | An attacker may flood the Flask server with requests, crashing or slowing down the service. |
| **Code injection or buffer overflow** (if input is not sanitized) | Even though it's not directly vulnerable now, any extension to file upload or filename-based input must validate properly. |

## 3. **Backdoors**

A **backdoor** is a hidden or unintended method of bypassing normal authentication or encryption.

Currently, there’s no explicit backdoor, but potential **unintentional backdoors** include:

| **Backdoor Type** | **Explanation** |
| --- | --- |
| **Key reuse without user consent** | If the same key is reused and not changed, it acts like a backdoor for attackers who already have the key. |
| **Debug print statements** | If the system logs keys and IVs in plain text, any attacker with access to logs can decrypt all data. |
| **Missing input validation** | If expanded to accept file names, poor input validation might allow path traversal attacks like ../../etc/passwd. |
| **verify=False disabling SSL** | This is like leaving a door open for an attacker to insert themselves between client and server and modify traffic. |

# **Mitigation Techniques**

**1. Use HTTPS (Secure Communication)**

* **Problem:** The client disables SSL verification using verify=False, which allows Man-in-the-Middle (MITM) attacks.
* **Mitigation:**
  + Use a valid SSL certificate on the Flask server.
  + Enforce verify=True in the client to ensure secure communication.

**2. Authentication and Authorization**

* **Problem:** The API endpoint (/upload-key) is publicly accessible without authentication.
* **Mitigation:**
  + Implement API key, token-based, or JWT (JSON Web Token) authentication.
  + Ensure only authenticated users can access endpoints.

**3. Key and IV Protection**

* **Problem:** Keys and IVs are exposed via print statements or stored insecurely.
* **Mitigation:**
  + Never print or log keys in plaintext.
  + Store keys securely using:
    - Environment variables
    - Encrypted databases (e.g., Vault, KeyStore)
    - File system with access control

**4. Input Validation and Sanitization**

* **Problem:** User inputs (in file uploads or JSON data) can be used for injection or traversal attacks.
* **Mitigation:**
  + Validate all JSON fields for type and length.
  + If filenames or file paths are taken from user, sanitize to prevent ../ path traversal.

**5. File Integrity Verification**

* **Problem:** No mechanism to verify if the file is tampered with after encryption.
* **Mitigation:**
  + Use hash-based verification (e.g., SHA-256):
    - Generate a hash of the file before encryption.
    - Verify the hash after decryption to ensure integrity.

**6. Limit and Monitor API Usage**

* **Problem:** APIs are vulnerable to abuse (e.g., brute-force, DoS).
* **Mitigation:**
  + Use rate limiting (e.g., Flask-Limiter) to limit the number of requests per user/IP.
  + Monitor server logs and enable IP blocking for malicious behavior.

# **Overall Description Of The Project**

**Title:**

**Secure File Encryption and Key Transmission System Using AES and Flask**

**Introduction**

In the digital age, data security is a critical concern due to the increasing threats of cyberattacks, data leaks, and unauthorized access. This project aims to develop a **secure file encryption system** that ensures **confidentiality** and **secure key exchange** using industry-standard encryption methods and web technologies.

The project focuses on two main functionalities:

1. **AES-based file encryption** on the client-side.
2. **Secure transmission of encryption keys** to a server using a **Flask-based REST API**.

**Objective**

The main goal of this project is to implement a simple yet secure mechanism to:

* Encrypt files using **AES (Advanced Encryption Standard)**.
* Transmit the **secret key** and **initialization vector (IV)** securely to a backend server.
* Ensure that the encryption process is safe from tampering, leakage, or interception.

**Technologies Used**

* **Python** for client-side and server-side development.
* **Flask** to create the backend REST API.
* **PyCryptodome** library for AES encryption.
* **Base64 Encoding** for safe transmission of binary data.
* **Postman** for API testing (optional).
* **Wireshark** for traffic analysis and verification (optional).

**System Workflow**

1. **Client-Side Encryption:**
   * A file is read in binary format.
   * AES key and IV are generated randomly.
   * The file is encrypted using AES in CBC mode.
   * The AES key and IV are encoded using Base64 for safe transmission.
2. **Key Upload to Server:**
   * A POST request is made to the Flask server.
   * JSON payload includes the Base64-encoded key and IV.
   * The server receives and prints the key and IV (in real-world use, it would store them securely).

**Key Features**

* **Secure File Encryption** using AES-128 or AES-256 bit symmetric encryption.
* **Flask API for Key Upload** enables separation of encryption and key management.
* **Base64 Encoding** allows binary keys to be sent over HTTP safely.
* **Flexible Design** allows future integration with file upload, decryption, and authentication mechanisms.

**Use Cases**

* Encrypting sensitive files before uploading to cloud storage.
* Secure communication between two systems sharing a secret file.
* Demonstrating basic principles of secure key exchange in a client-server model.

**Security Aspects Implemented**

* AES encryption with random IV to prevent pattern leakage.
* Secure key transmission via HTTP (can be upgraded to HTTPS).
* Input validation and structured JSON communication.
* Logging and error handling for robustness.
* Best practices recommended for production deployment (e.g., SSL, API authentication).

**Learning Outcomes**

* Practical knowledge of **symmetric encryption (AES)**.
* Building and testing **REST APIs with Flask**.
* Secure data exchange principles (Base64, SSL, authentication).
* Real-world concepts like **key management**, **encryption modes**, and **secure file handling**.

# **Implementation**

**1. AES Encryption/Decryption**

AES (Advanced Encryption Standard) is a symmetric block cipher that encrypts data in blocks of 16 bytes. We use **AES-128 CBC mode** for this implementation.

**Key Steps:**

* **Key and IV Generation**:  
  The client generates a **128-bit key** (16 bytes) and a **128-bit IV** using Crypto.Random.get\_random\_bytes(). This ensures the randomness and strength of the encryption.
* **Padding**:  
  Since AES requires data in 16-byte blocks, input data is padded using a **PKCS#7-style** padding scheme before encryption. This involves adding n bytes, each having the value n, where n is the number of padding bytes needed.
* **Encryption Process**:
  + The file is read in binary mode.
  + The data is padded and encrypted using the key and IV in **AES.MODE\_CBC** mode.
  + The encrypted data is saved as a new file with a .enc extension.
* **Decryption Process**:
  + The encrypted file is read.
  + AES decryption is performed using the same key and IV.
  + Padding is removed to retrieve the original data.
  + The output is saved as a new file (e.g., decrypted\_sample.txt).

**2. Flask API for Key Exchange**

To simulate secure key transmission, a **Flask-based HTTPS server** is implemented. The server exposes a **/upload-key endpoint** which accepts POST requests containing the AES key and IV in base64 format.

**Key Steps:**

* **API Route /upload-key**:
  + The client sends a JSON object containing the base64-encoded key and iv.
  + The server prints/stores the received values for future decryption.
  + The Flask app is run with SSL enabled using a **self-signed certificate** (cert.pem, key.pem) to provide an HTTPS interface.
* **HTTPS Security**:
  + The API is served over HTTPS using ssl\_context=('cert.pem', 'key.pem').
  + This ensures the confidentiality and integrity of the key transmission.

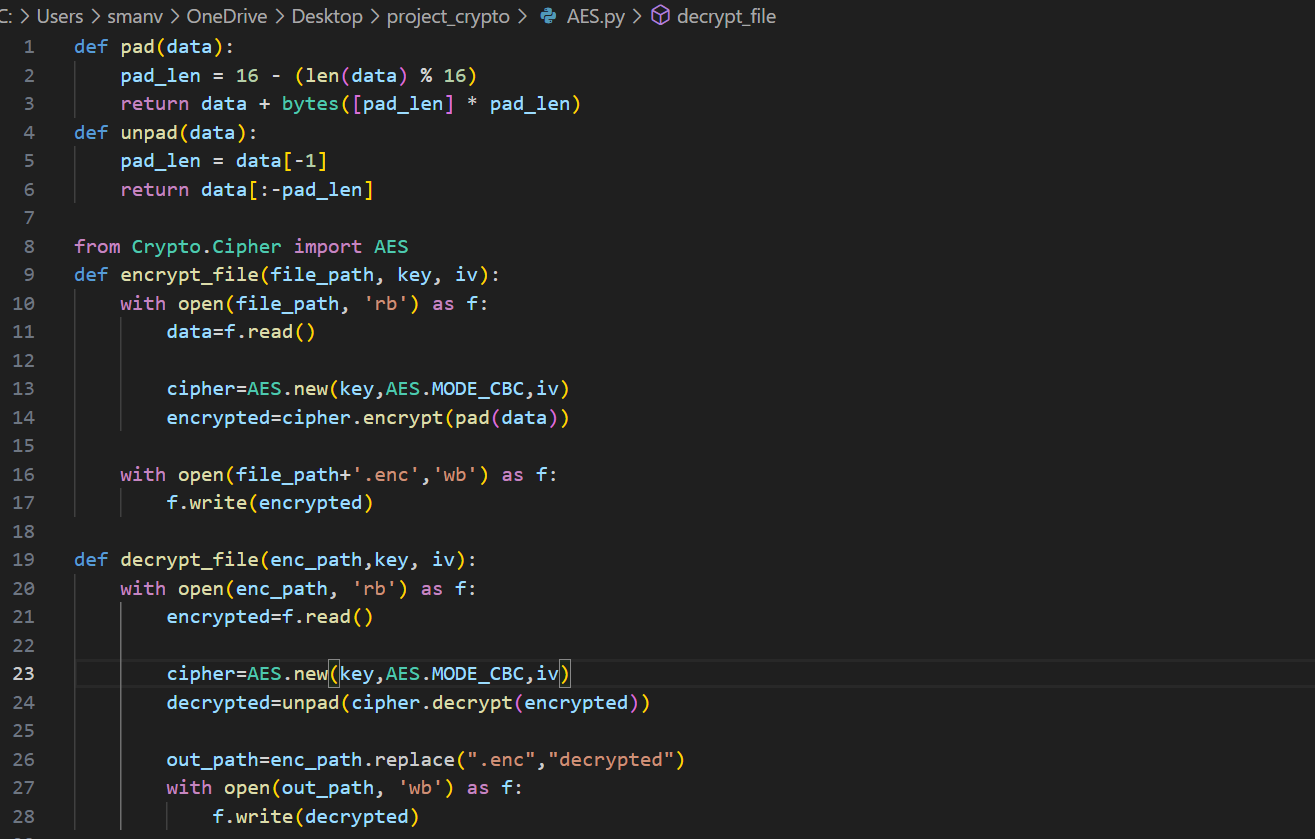
**3. Client–Server Interaction**

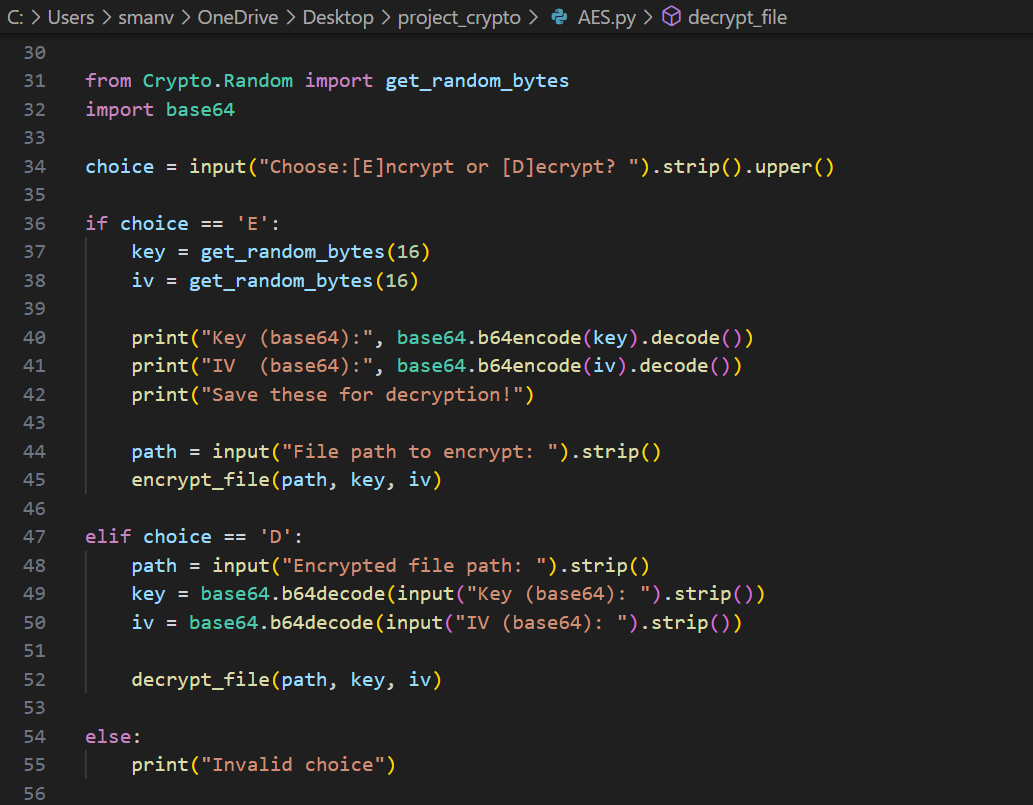
**Data Flow:**

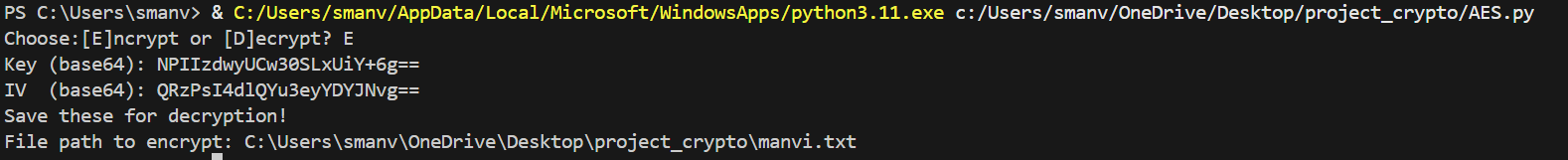
1. **Encryption Mode (Client)**:
   * User selects a file for encryption.
   * The client generates the key and IV.
   * The file is encrypted and saved.
   * The key and IV are encoded in base64 and sent to the Flask server over HTTPS.
2. **Decryption Mode (Client)**:
   * User provides the base64 key and IV received earlier.
   * The client decrypts the .enc file using these values.
   * The decrypted output is saved to disk.

**Screenshots**

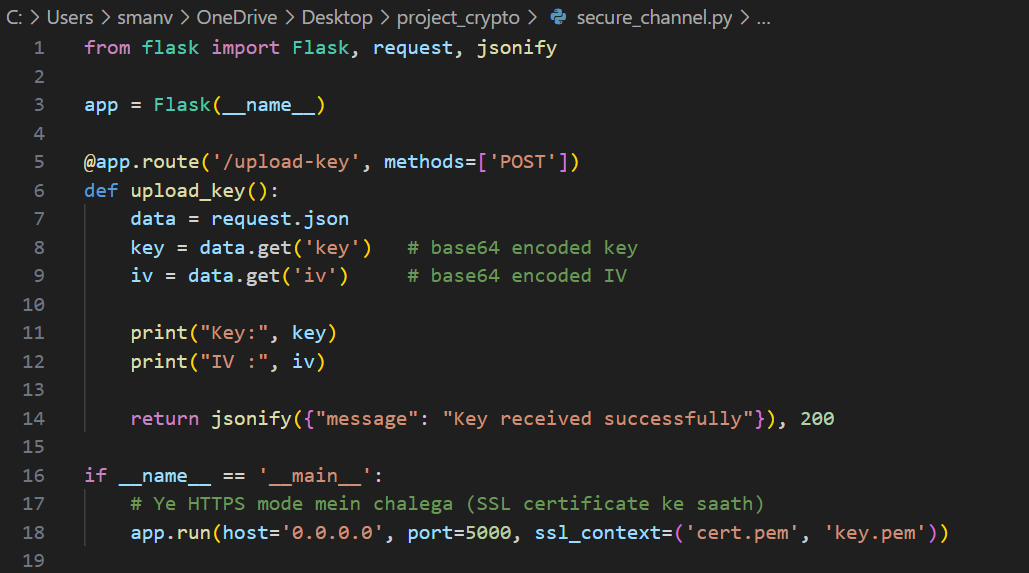
* 1. AES:

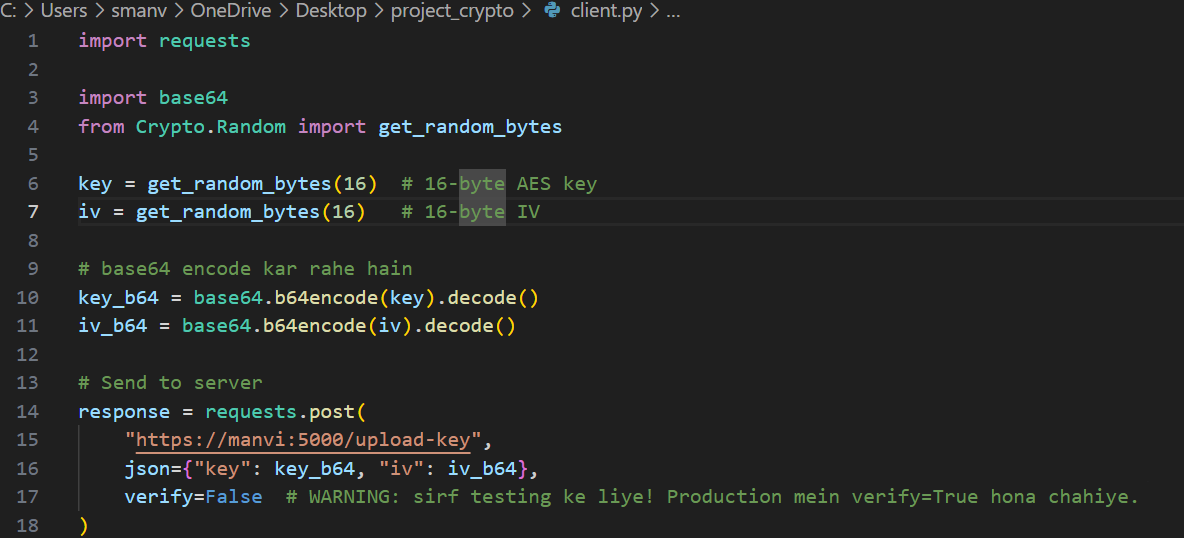


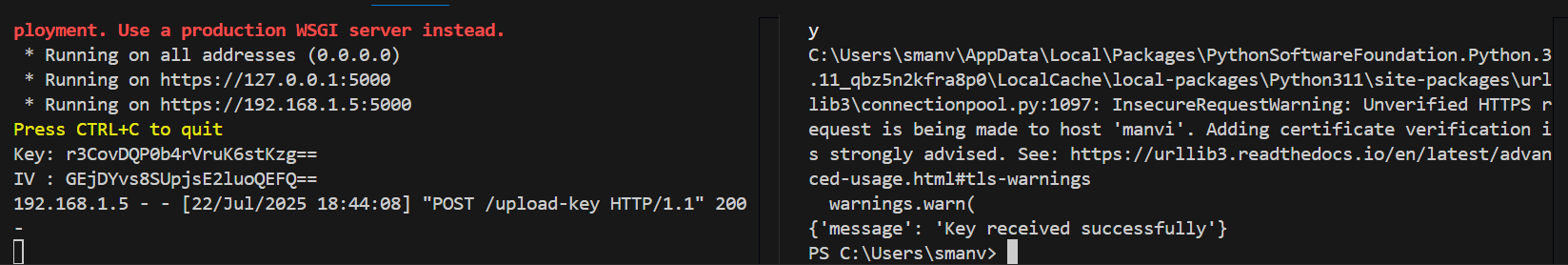




* 1. **Flask API for Key Exchange**







# **ALGORITHM 2: CIPHER CAESER**

**Introduction to Caesar Cipher**

The **Caesar Cipher** is one of the most well-known and earliest encryption techniques in classical cryptography. It was used by **Julius Caesar**, a Roman military general and leader, to protect military messages from being intercepted by enemies. This cipher is a form of **substitution cipher**, where each letter in the plaintext is shifted a fixed number of places down or up the alphabet.

**Working Principle**

In a Caesar Cipher, each alphabet in the original message (plaintext) is replaced by another letter a fixed number of positions away in the alphabet. This fixed number is known as the **key**. For example, with a key of 3:

* A → D
* B → E
* C → F
* ...
* X → A
* Y → B
* Z → C

So the message **"HELLO"** becomes **"KHOOR"** after encryption with a shift of 3.

**Mathematical Formula**

Let:

* **P** = position of the original letter (A = 0, B = 1, ..., Z = 25)
* **K** = key (number of positions to shift)
* **C** = position of the encrypted letter

Then,

* **Encryption**:  C = (P + K) mod 26
* **Decryption**:  P = (C - K + 26) mod 26

This ensures the result wraps around if it goes past 'Z'.

**Features**

* Simple to implement and understand.
* Works with alphabetic characters.
* Key-based transformation.

**Limitations**

* **Low Security**: Easily breakable using brute-force (only 25 possible keys).
* **Not suitable for modern encryption**.
* Vulnerable to **frequency analysis attacks**.

# **Vulnerabilities, Threats, and Backdoors in Caesar Cipher**

The Caesar Cipher is a **classical substitution cipher** and one of the most basic forms of encryption. While it is useful for educational and demonstration purposes, it suffers from several significant security flaws that make it unsuitable for real-world secure communication.

**1. Vulnerabilities**

| **Vulnerability** | **Description** |
| --- | --- |
| **Low Key Space** | Caesar Cipher only has **25 possible shifts** (ignoring 0), which means brute force attack is trivial. |
| **No Key Management** | The key (shift value) is often hardcoded or transmitted insecurely. |
| **Pattern Leakage** | Since letter substitution is consistent, frequency analysis can easily reveal common letters like 'E', 'T', etc. |
| **No Data Integrity** | It provides no way to check if the ciphertext has been altered. |
| **Fixed Character Mapping** | Every time 'A' is encrypted, it becomes the same character — makes it predictable. |

**2. Potential Threats**

| **Threats** | **Impact** |
| --- | --- |
| **Brute Force Attack** | An attacker can try all 25 shifts and recover the plaintext quickly. |
| **Frequency Analysis Attack** | By analyzing letter frequencies, an attacker can decode text even without knowing the shift value. |
| **Known Plaintext Attack** | If an attacker has a known sample of plaintext and ciphertext, the key can be immediately derived. |
| **Replay Attack** | If Caesar Cipher is used in communication without timestamps or freshness indicators, old messages can be reused. |
| **Key Leakage** | If the shift value is sent in plaintext (or guessed), encryption becomes meaningless. |

**3. Possible Backdoors**

Although Caesar Cipher is too simple to have sophisticated backdoors like in modern cryptosystems, the following can be considered **backdoor-like weaknesses**:

| **Backdoor Type** | **Explanation** |
| --- | --- |
| **Hardcoded Shift** | If the shift value (e.g., 3) is fixed and known, anyone can decrypt messages without access control. |
| **Predictable Output** | Same input always leads to same output → easy pattern recognition and reverse-engineering. |
| **No Salting or Randomness** | Lack of randomness in encryption means same message encrypted twice will always look the same. |
| **Insider Knowledge** | If someone knows the implementation and key logic, they can build tools to decrypt messages instantly. |

# 

# **Mitigation Techniques**

**1. Replace with Stronger Encryption**

| **Technique** | **Explanation** |
| --- | --- |
| **Use AES or RSA** | Switch to **Advanced Encryption Standard (AES)** for symmetric encryption or **RSA** for asymmetric encryption. |
| **Use Modern Libraries** | Use secure libraries like PyCryptodome, cryptography, or OpenSSL for proper encryption standards. |

**2. Add Randomness**

| **Technique** | **Effect** |
| --- | --- |
| **Use Salt or IV (Initialization Vector)** | Add a random number to the message or shift to ensure unique ciphertexts for same plaintexts. |
| **Non-deterministic Shifting** | Introduce random factors during encryption to break repeatable patterns. |

**📉 3. Frequency Flattening Techniques**

| **Technique** | **Purpose** |
| --- | --- |
| **Homophonic Substitution** | Map one character to multiple symbols randomly to defeat frequency analysis. |
| **Transposition + Substitution** | Use a hybrid method to make ciphertext less predictable. |

**4. Improve Key Management**

| **Technique** | **Benefit** |
| --- | --- |
| **Secure Key Exchange** | Use Diffie-Hellman or RSA to securely exchange keys. |
| **Store Keys Safely** | Never hardcode keys; use secure key storage or environment variables. |

**5. Add Authentication & Integrity**

| **Feature** | **Use** |
| --- | --- |
| **MAC (Message Authentication Code)** | Ensures message was not tampered with. |
| **HMAC (Hashed MAC)** | Combines cryptographic hash + key to verify authenticity. |
| **Digital Signatures** | Verify sender identity in asymmetric encryption. |

# **Implementation**

This section explains the step-by-step implementation of the Caesar Cipher algorithm in Python, along with enhancements for digit and special character handling.

**🔹 1. Function: encrypt\_caesar(text, shift)**

**Purpose**:  
To encrypt a given plaintext using the Caesar Cipher technique with a fixed shift value.

**Steps**:

* Initialize:
  + m → empty string to build the encrypted result.
  + n → list to store original digits.
* Loop through each character in the input text:
  + If the character is **an uppercase letter**:
    - Convert it to ASCII using ord(), apply shift, and wrap around using % 26.
    - Convert it back using chr() and append to m.
  + If it is a **lowercase letter**:
    - Same logic, but adjust for lowercase ASCII values.
  + If it is a **digit**:
    - Append the original digit to n.
    - Append '0' in place of the digit in m.
  + If it is a **special character or space**:
    - Append it unchanged to m.
* Return:
  + Encrypted string m
  + List of original digits n

**🔹 2. Function: decrypt\_caesar(text, shift, n)**

**Purpose**:  
To decrypt the encrypted text using the Caesar Cipher logic and reinsert digits in the correct positions.

**Steps**:

* Initialize:
  + q → decrypted result string
  + digit\_index → index counter for digits from list n
* Loop through each character in text:
  + If it is an **uppercase or lowercase letter**:
    - Apply reverse shift using ord() - shift, wrap using % 26, and convert back to character.
  + If the character is '0':
    - Replace it with the digit at current digit\_index from list n.
    - Increment digit\_index.
  + If it is a **special character**:
    - Keep it unchanged.
* Return the final **decrypted** text q.

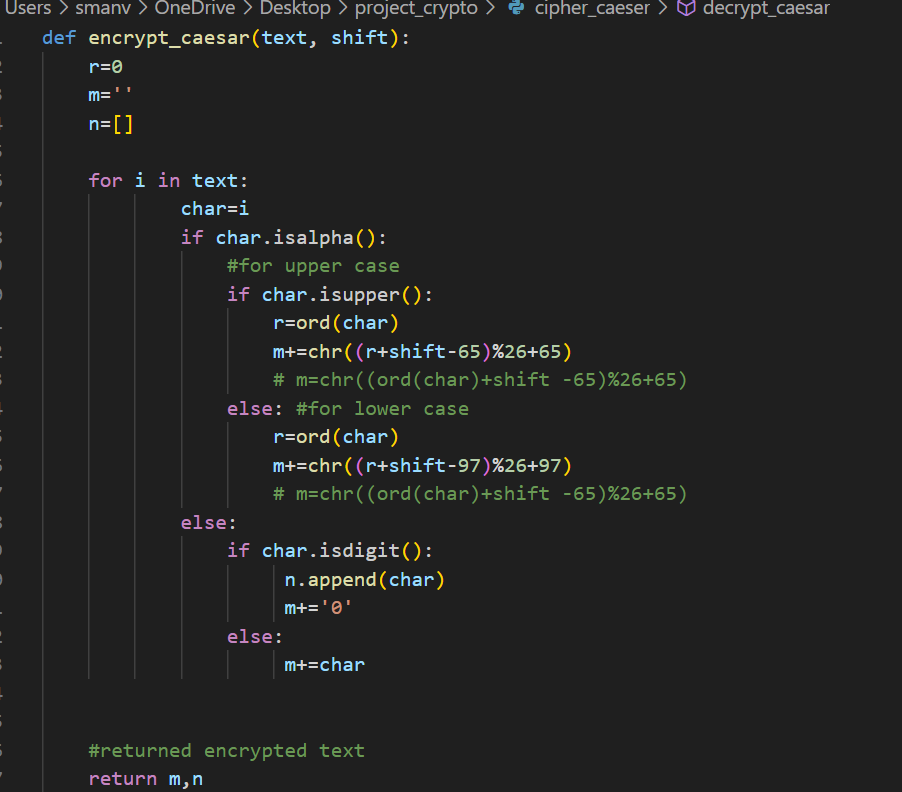
**🔹 3. Main Execution Block**

**Purpose**:  
To collect input from the user, call encryption and decryption functions, and display results.

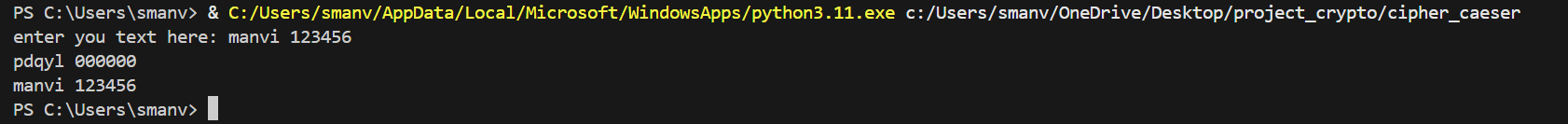
**Steps**:

1. Ask the user for the **input text**.
2. Set a **fixed shift** value (shift = 3).
3. Call the encrypt\_caesar() function → receive encrypted text and digit list.
4. Call the decrypt\_caesar() function → receive decrypted original text.

# **Screenshots**







# **Problem Statement**

In today's digital world, transmitting sensitive data over the internet poses serious risks due to the increasing number of cyber-attacks, data breaches, and unauthorized access. Traditional data transfer methods often lack strong security mechanisms, making them vulnerable to interception, tampering, or leakage of confidential information. There is a pressing need for a secure, reliable, and easy-to-implement solution that ensures the confidentiality and integrity of data during transmission. This project addresses that problem by designing a secure communication system using AES (Advanced Encryption Standard) encryption for data protection and a Flask-based API for secure client-server interaction. The goal is to protect sensitive data from being exposed or altered while being transmitted over the network.

# **Limitations**

While the project successfully demonstrates secure communication using AES encryption and Flask API, it has several limitations that must be acknowledged:

1. **Key Exchange Security**: AES is a symmetric encryption algorithm, meaning the same key is used for both encryption and decryption. However, securely sharing this key between the client and server is not handled in this project. If the key is intercepted during transmission, the entire system becomes vulnerable. A more secure method like RSA (asymmetric encryption) or a secure key exchange protocol like Diffie-Hellman should be integrated but is currently missing.
2. **Lack of Authentication and Authorization**: The Flask API currently does not implement any authentication mechanisms like API keys, tokens, or user sessions. This means that anyone who knows the endpoint can potentially send requests to the server, making the system vulnerable to misuse or attacks like spoofing and unauthorized access.
3. **No HTTPS by Default**: The Flask development server used in the project does not support HTTPS out-of-the-box. Without HTTPS, data—including keys and IVs—can be exposed in plaintext during transmission, especially on unsecured networks.
4. **Limited Error Handling and Logging**: The system lacks proper exception handling and logging mechanisms. Any unexpected error (e.g., incorrect data format, missing key, etc.) can crash the server or make debugging difficult, reducing reliability.
5. **No Data Integrity Check**: While AES ensures confidentiality, the project does not implement message authentication codes (MACs) or hashes to verify if the data has been tampered with during transmission. Without integrity checks, an attacker could potentially alter the ciphertext without detection.
6. **Manual IV Management**: The Initialization Vector (IV) is randomly generated but sent alongside the key. If not handled securely, this may allow certain cryptographic attacks (like replay attacks) depending on implementation. Best practices like using a new IV per encryption and securing its transmission are not strictly enforced.
7. **No Front-End or User Interface**: The project operates via backend scripts without a graphical or web-based interface. This limits usability for non-technical users and requires manual script execution for sending and receiving data.
8. **Platform Dependency**: The encryption/decryption and API communication depend on Python scripts and libraries (like pycryptodome and requests), which might not be cross-platform friendly or suitable for embedded/low-power environments without adjustments.
9. **Testing Limitations**: The current setup is designed for a controlled environment and has not been stress-tested under real-world conditions such as packet loss, network delay, or man-in-the-middle attack simulations.

Top of Form

Bottom of Form

# **CONCLUSION**

The Caesar Cipher is a foundational technique in the field of cryptography, demonstrating how simple substitution-based encryption can provide basic confidentiality. Through this project, we successfully implemented both the **encryption** and **decryption** processes of the Caesar Cipher using a fixed key (shift value), showing how data can be transformed into an unreadable form and then accurately restored.

Although Caesar Cipher is not suitable for modern-day security due to its simplicity and vulnerability to brute-force attacks, it plays a significant role in helping understand the **core principles of encryption**, such as key usage, character shifting, and handling non-alphabetic data.

This project also highlights the importance of understanding **data security**, **threats**, and the **need for stronger cryptographic techniques** in today's digital world.

The AES (Advanced Encryption Standard) algorithm is one of the most powerful and widely adopted symmetric-key encryption techniques used for securing sensitive data. In this project, we implemented AES encryption using Python, where we generated a **random 128-bit key and IV (Initialization Vector)**, encoded them using **Base64**, and securely transmitted them to a server via a **Flask API**.

Through this implementation, we demonstrated how AES can be integrated into real-world applications for **file encryption**, **secure communication**, and **key exchange**. We also highlighted how critical it is to manage encryption keys and IVs securely to ensure data confidentiality and integrity.

This project allowed us to:

* Understand the **block-based encryption mechanism** of AES.
* Implement **secure communication** between client and server using encryption.
* Learn about **threats and vulnerabilities**, such as key exposure or insecure transmission.

Although AES is very strong cryptographically, its effectiveness depends heavily on secure key management, secure transmission practices, and proper implementation.

# **References**

**For AES (Advanced Encryption Standard):**

1. **NIST FIPS 197 PDF (Reference #2)**  
    This is the **official document** where AES was standardized.
   * **Issued by:** U.S. National Institute of Standards and Technology (NIST)
   * **Direct Link:** https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf  
     **Authentic Government Source**
2. **The Design of Rijndael (Reference #1)**  
    Written by the **creators of AES** (Joan Daemen & Vincent Rijmen).
   * Publisher: **Springer**
   * This book explains the math and logic behind AES.  
     **Academic Book, widely cited in cryptography literature**
3. **Stallings Textbook (Reference #3)**  
    Used in many university-level courses for cybersecurity and cryptography.  
    **Trusted source in computer science education**
4. **PyCryptodome & Cryptography Libraries (References #4 & #5)**  
    These are the **real Python libraries** used for encryption/decryption.
   * Links go to their official documentation  
     **Used in professional and academic Python projects**
5. **OWASP Cryptographic Cheat Sheet (Reference #8)**  
   OWASP (Open Web Application Security Project) is a **globally respected authority** on security best practices.  
   **Ideal for understanding how to securely manage encryption**

**For Caesar Cipher:**

**GeeksforGeeks (Reference #6)**  
While more beginner-friendly, it's still a valid source for simple algorithms like Caesar Cipher.  
Widely used for explanations and student projects

**For Flask API Usage:**

1. **RealPython (Reference #7)**  
   A reputable site for learning Python professionally  
   Good for learning Flask-based API implementation (used in your AES project)