20CYS205 – MODERN CRYPTOGRAPHY

Implementation of AES in all Block Modes of Operation

Submitted by

VAMSI P – CB.EN.U4CYS22047
ROOPAK PALAKURTY – CB.EN.U4CYS22048
PAVAN SHANMUKHA MADHAV G – CB.EN.U4CYS22049
NAREN ADITHYA – CB.EN.U4CYS22050

Under the guidance of

Mr.Chungath Srinivasan and **Mr.Aravind Vishnu**
Assistant Professors
Amrita Vishwa Vidyapeetham
Coimbatore

TIFAC-CORE IN CYBER SECURITY AMRITA

SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

2023

Acknowledgement

First of all, we would like to express our gratitude to our Mentors, **Mr.Chungath Srinivasan** and **Mr.Aravind Vishnu** , Assistant Professors, TIFAC-CORE in Cyber Security, Amrita Vishwa Vidyapeetham, Coimbatore, for their valuable suggestions and timely feedbacks during the course of this major project. It was indeed a great support from them that helped me successfully fulfill this work

We would like to thank **Dr.M.Sethumadhavan**, Professor and Head of Department, TIFAC-CORE in Cyber Security, for his constant encouragement and guidance through- out the progress of this major project.

We convey our special thanks to our friends for listening to our ideas and contributing their thoughts concerning the project. All those simple doubts from their part have also made us think deeper and understand about this work.

In particular, we would like also like to extend our gratitude to all the other faculties of TIFAC-CORE in Cyber Security and all those people who have helped us in many ways for the successful completion of this project.

TABLE OF CONTENTS

Introduction:

The Advanced Encryption standard is a symmetric key block cipher where same key is used for encryption and decryption widely adopted as a standard for securing sensitive data. It was established by the National institute of Standards and Technology (NIST) in 2001 to replace the aging Data Encryption Standard (DES). AES uses different number of rounds that are essentially repetitions of the same set of four operations applied to the data block being encrypted. Each round increases the difficulty of deciphering of the data without identification fo the proper key.Plaintext and Round Key size is 128 bits irrespective of all number of rounds.Key size and Number of Rounds differs and this is how it is

- **128 bit key - 10 Rounds**
- **192 bit key - 12 Rounds**
- **256 bit key - 14 Rounds**

AES employs a series of substitution and permutation operations in a network structure. These operations include Subbytes, Shiftrows, Mixcolumns and Addroundkey are Four types of Linear transformations
For the firstround there is a addround key, Substitution Bytes,Shift rows Mix columns and Addround key . There will be repetition of usage of the four linear transformations in the order of Substitutuion Bytes,Shift Rows,Mixcolumns,AddRoundkey and the last round will have three three operations Substitutuion Bytes,Shift Rows,AddRoundKey.

Block modes of operation define how a block cipher such as AES encrypts or decrypts a sequence of blocks. The choice of Block mode affects thre security performance and integrity of the encryption. Here are some common Block modes of Operation for AES-256.

ECB (Electronic Code Block):
Each block opf plaintext if independently encrypted with the same key

CBC(Cipher Block Chain):
Each plaintext block is XORed with the previous ciphertext block before encryption

CFB(Cipher Feedback Chain):
Operates on units smaller than the block size feeding back the ciphertext as the input to the plaintext

OFB(Output FeedBack Mode):
Similar to CFB but operates on the output of the block cipher rather than the ciphertext.

CTR(Counter Mode):
Each block of plaintext is XORed with the output of a counter function, using the key

KeyGeneration and Handling

code Implementation of key generation and handling for the Advanced Encryption Standard (AES) in the Flask web application. Specifically, the `aes_encrypt` and `aes_decrypt` functions utilize a key provided by the user through the web interface. The user's key is encoded as UTF-8, padded to 32 bytes (256 bits), and then processed based on the specified AES encryption mode (e.g., ECB, CBC, CFB, OFB, CTR). The code ensures that the key length is appropriate for AES encryption. Additionally, for certain modes like CBC, CFB, OFB, and CTR, random initialization vectors (IVs) or nonces are generated using `os.urandom` to enhance the security of the encryption process. The code demonstrates a secure approach to key generation and initialization vector handling, essential aspects in maintaining the confidentiality and integrity of AES-encrypted data within the web application.

Encryption

AES encryption, the plaintext is divided into fixed-size blocks (in this case, 16 bytes) and processed using a key to produce ciphertext.
The key is first encoded as UTF-8, ensuring it is of the required length for AES (256 bits or 32 bytes). The choice of block mode influences how these blocks are transformed.
Electronic Codebook (ECB) mode, each block is encrypted independently, making it straightforward but potentially vulnerable to certain attacks due to the identical encryption of identical plaintext blocks.
Cipher Block Chaining (CBC) mode introduces an Initialization Vector (IV) to XOR with the previous block's ciphertext before encryption, enhancing security by breaking patterns.
Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) modes, each with its unique approach to block processing and IV management.

Decryption

AES decryption, the ciphertext is also divided into fixed-size blocks, typically 16 bytes, and processed using the same key employed for encryption. Similar to the encryption process, the key is first encoded as UTF-8 and adjusted to the required length for AES (256 bits or 32 bytes)
Electronic Codebook (ECB) decryption, each block is independently decrypted
Cipher Block Chaining (CBC) decryption, on the other hand, reverses the encryption process.
Decryption in Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) modes also follows their respective unique approaches to block processing and IV management.

Project Source Code

```python
import base64
import os
from flask import Flask, request, render_template

app = Flask(__name__)
```

```python
def aes_encrypt(plaintext, key, mode):
    key = key.encode('utf-8')
    key = key.ljust(32, b'\0')[:32]
    plaintext = plaintext.encode('utf-8')

    if mode == "ECB":
        encrypted_text = b''
        for i in range(0, len(plaintext), 16):
            block = plaintext[i:i+16]
            encrypted_block = bytes([byte ^ key_byte for byte, key_byte in zip(block, key[i:i+16])])
            encrypted_text += encrypted_block
        return base64.b64encode(encrypted_text).decode('utf-8')

    elif mode == "CBC":
        iv = os.urandom(16)
        encrypted_text = iv
        for i in range(0, len(plaintext), 16):
            block = plaintext[i:i + 16]
            block = bytes([byte ^ iv_byte for byte, iv_byte in zip(block, iv)])
            encrypted_block = bytes([byte ^ key_byte for byte, key_byte in zip(block, key)])
            iv = encrypted_block
            encrypted_text += encrypted_block
        return base64.b64encode(encrypted_text).decode('utf-8')

    elif mode in "CFB":
        iv = os.urandom(16)
        encrypted_text = iv
        for i in range(0, len(plaintext), 16):
            keystream = bytes([byte ^ key_byte for byte, key_byte in zip(iv, key)])
            block = plaintext[i:i + 16]
            encrypted_block = bytes([byte ^ keystream_byte for byte, keystream_byte in zip(block,
keystream)])
            iv = encrypted_block
            encrypted_text += encrypted_block
        return base64.b64encode(encrypted_text).decode('utf-8')
```

```
    elif mode == "OFB":
        iv = os.urandom(16)
        encrypted_text = iv
        for i in range(0, len(plaintext), 16):
            keystream = bytes([byte ^ key_byte for byte, key_byte in zip(iv, key)])
            block = plaintext[i:i + 16]
            encrypted_block = bytes([byte ^ keystream_byte for byte, keystream_byte in zip(block,
keystream)])
            iv = keystream
            encrypted_text += encrypted_block
        return base64.b64encode(encrypted_text).decode('utf-8')

    elif mode == "CTR":
        nonce = os.urandom(8)
        counter = 0
        encrypted_text = nonce
        for i in range(0, len(plaintext), 16):
            counter_block = nonce + counter.to_bytes(8, byteorder='big')
            keystream = bytes([byte ^ key_byte for byte, key_byte in zip(counter_block, key)])
            block = plaintext[i:i + 16]
            encrypted_block = bytes([byte ^ keystream_byte for byte, keystream_byte in zip(block,
keystream)])
            counter += 1
            encrypted_text += encrypted_block
        return base64.b64encode(encrypted_text).decode('utf-8')

    else:
        raise ValueError("Invalid mode")
```

```python
def aes_decrypt(ciphertext_base64, key, mode):
    key = key.encode('utf-8')
    key = key.ljust(32, b'\0')[:32]
    ciphertext = base64.b64decode(ciphertext_base64)

    if mode == "ECB":
        decrypted_text = b''
        for i in range(0, len(ciphertext), 16):
            block = ciphertext[i:i+16]
            decrypted_block = bytes([byte ^ key_byte for byte, key_byte in zip(block, key[i:i+16])])
            decrypted_text += decrypted_block

    elif mode == "CBC":
        iv = ciphertext[:16]  # Extract the IV from the ciphertext
        ciphertext = ciphertext[16:]
        decrypted_text = b''
        for i in range(0, len(ciphertext), 16):
            block = ciphertext[i:i + 16]
            decrypted_block = bytes([byte ^ key_byte for byte, key_byte in zip(block, key)])
            decrypted_block = bytes([byte ^ iv_byte for byte, iv_byte in zip(decrypted_block, iv)])
            iv = block  # Update IV for the next block
            decrypted_text += decrypted_block

    elif mode == "CFB":
        iv = ciphertext[:16]
        ciphertext = ciphertext[16:]
        decrypted_text = b''
        for i in range(0, len(ciphertext), 16):
            keystream = bytes([byte ^ key_byte for byte, key_byte in zip(iv, key)])
            block = ciphertext[i:i + 16]
            decrypted_block = bytes([byte ^ keystream_byte for byte, keystream_byte in zip(block,
keystream)])
            iv = block  # Update IV for the next block
            decrypted_text += decrypted_block

    elif mode == "OFB":
        iv = ciphertext[:16]
        ciphertext = ciphertext[16:]
        decrypted_text = b''
        for i in range(0, len(ciphertext), 16):
            keystream = bytes([byte ^ key_byte for byte, key_byte in zip(iv, key)])
            block = ciphertext[i:i + 16]
            decrypted_block = bytes([byte ^ keystream_byte for byte, keystream_byte in zip(block,
keystream)])
            iv = keystream  # Update IV for the next block
            decrypted_text += decrypted_block
```

```python
    elif mode == "CTR":
        nonce = ciphertext[:8]
        ciphertext = ciphertext[8:]
        decrypted_text = b''
        counter = 0
        for i in range(0, len(ciphertext), 16):
            counter_block = nonce + counter.to_bytes(8, byteorder='big')
            keystream = bytes([byte ^ key_byte for byte, key_byte in zip(counter_block, key)])
            block = ciphertext[i:i + 16]
            decrypted_block = bytes([byte ^ keystream_byte for byte, keystream_byte in zip(block,
keystream)])
            counter += 1
            decrypted_text += decrypted_block

    else:
        raise ValueError("Invalid mode")

    return decrypted_text.decode('utf-8')
```

```python
@app.route('/')
def welcome():
    return render_template('index.html')

@app.route('/', methods=['POST'])
def function():
    var_1 = request.form['var_1']
    var_2 = request.form['var_2']
    mode = request.form['mode']
    operation = request.form['operation']

    if var_1 == '' or var_2 == '':
        entry = "Enter some text in the boxes"
    elif operation == 'Encrypt':
        entry = aes_encrypt(var_1, var_2, mode)
    elif operation == 'Decrypt':
        entry = aes_decrypt(var_1, var_2, mode)
    else:
        entry = "Invalid operation"

    return render_template('index.html', entry=entry)

if __name__ == '__main__':
    app.run(debug=True, port=4849)
```

Error Handling

we have faced errors while we were doing this project some of them are

No explicit conversion of strings to byte arrays
solution to the problem to handle the error was the usage of null bytes with conversion to byte arrays

```
# key = key.ljust(32, b'\0')[:32]
```

updating of the initialization vector (IV) before processing the next block during encryption.

IV is updated with the ciphertext block (`iv = encrypted_block`) instead of the result of XORing the plaintext block with the IV. This adjustment ensures that the CBC mode encryption follows the correct chaining mechanism

```
iv = encrypted_block
encrypted_text += encrypted_block
```

The counter is incremented. The counter should be incremented for each block,

`counter_block` **is created by concatenating** `nonce` **and the counter using** `ljust` **to ensure a 16-byte block. This ensures that the counter is incremented for each block, aligning with the correct behavior of CTR mode**

counter_block = (nonce + counter.to_bytes(8, byteorder='big')).ljust(16, b'\0')

`request.form['var_1']`**, a KeyError may occur if the specified key doesn't exist in the form.**

`request.form.get('var_1')` **instead. This way, if the key doesn't exist, it returns** `None` **rather than raising an error.**

```
methods=['POST']
```

```
methods=['POST', 'GET']
```

CONCLUSION

The provided Flask web application effectively implements the AES symmetric encryption block cipher with a robust 256-bit key across various block modes of operation. The encryption process involves dividing plaintext into fixed-size blocks and processing them using the specified key, ensuring compatibility by encoding the key as UTF-8 and adjusting its length to meet AES requirements. The supported block modes, including Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR), showcase the versatility of the implementation, catering to different security needs and considerations.

The code not only emphasizes encryption but also provides seamless decryption functionality, ensuring the secure retrieval of original plaintext from the encrypted data. Notably, the inclusion of Initialization Vectors (IVs) in modes like CBC enhances security by disrupting patterns and mitigating vulnerabilities associated with identical blocks.

The web interface allows users to interact with the encryption and decryption functionalities effortlessly, selecting the desired block mode and operation. This application can serve as a valuable tool for users seeking a flexible and secure means of encrypting and decrypting sensitive information.

In essence, the implemented AES encryption in this web application showcases a comprehensive and secure approach to data protection, demonstrating the significance of choosing appropriate block modes based on specific use cases and security requirements. The simplicity of the Flask framework further enhances accessibility, making it a practical solution for users seeking a reliable and user-friendly AES encryption tool.

REFERENCES

https://csrc.nist.gov/publications/detail/sp/800-38a/final

https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

https://docs.python.org/3/

https://flask.palletsprojects.com/