CS 454 Final Project

Madeline Veric

November 22, 2023

1 AES Encryption and Decryption

1.1 Key Generation and Preparing a Message

To start off with AES, we first need to generate a key. In order to generate a key for AES encryption and decryption, we need to get a random byte string of length N where N is the number of bytes per block. In this case, we use 16 since the program follows an AES-128 bit structure. We can then make any message by taking a string literal and turn it into a sequence of octet bytes by applying the byte operator to it, which can be seen below.

```
# Replace get_random_bytes parameter with 16,24,or 32.
key = get_random_bytes(16)  # for AES-128 we use 16.

# Replace string with the message you want to encrypt
plaintext = b'Hello World!'
```

Figure 1: Key and Message Generation in Main

1.2 ECB Encryption and Decryption

ECB mode or Electronic Code-book is a AES mode where each block of plain-text is encrypted independently of any other block. It is semantically insecure due to its correlation between blocks. The encryption for this method expects data to have a length multiple of the block size (16 in this case due to our key being 16 bytes of length). We use ECB because it is simple but does not hide its pattern well and identical plain-text blocks produce identical cipher-text blocks.

```
def ecb_encrypt(plaintext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = cipher.encrypt(pad(plaintext, AES.block_size))
    return ciphertext

def ecb_decrypt(ciphertext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)
    return plaintext
```

Figure 2: Definition of ECB Encryption and Decryption Functions

```
# ECB Mode
ecb_ciphertext = ecb_encrypt(plaintext, key)
ecb_decrypted = ecb_decrypt(ecb_ciphertext, key)
print(f"ECB Decrypted: {ecb_decrypted.decode()}")
```

Figure 3: Use of ECB encryption and decryption functions in main.

1.3 CBC Encryption and Decryption

CBC mode or Cipher block Chaining is a AES mode where each block of plain-text is XORed with the previous cipher-text block before encryption. The only exception to this is when we originally use this mode and we need to read in an iv or Initialization Vector Value in order to start the encryption mode. It is mainly used to provide diffusion while makes identical plaint-text blocks produce different cipher-text blocks and offers better security overall compared to ECB.

```python
def cbc_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    block_size = AES.block_size

    previous_block = iv

    for i in range(0, len(plaintext), block_size):
        block = plaintext[i:i+block_size]

        # Padding the block before XOR
        padded_block = pad(block, block_size)

        xor_result = bytes(a ^ b for a, b in zip(padded_block, previous_block))
        encrypted_block = cipher.encrypt(xor_result)
        ciphertext += encrypted_block
        previous_block = encrypted_block

    return ciphertext

def cbc_decrypt(ciphertext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    decrypted_text = b""
    block_size = AES.block_size

    previous_block = iv

    for i in range(0, len(ciphertext), block_size):
        block = ciphertext[i:i+block_size]
        decrypted_block = cipher.decrypt(block)
        xor_result = bytes(a ^ b for a, b in zip(decrypted_block, previous_block))
        decrypted_text += xor_result
        previous_block = block

    return unpad(decrypted_text, block_size)
```

Figure 4: Definition of CBC Encryption and Decryption Functions

```python
# CBC Mode
iv_cbc = get_random_bytes(AES.block_size)
cbc_ciphertext = cbc_encrypt(plaintext, key, iv_cbc)
cbc_decrypted = cbc_decrypt(cbc_ciphertext, key, iv_cbc)
print(f"CBC Decrypted: {cbc_decrypted.decode()}")
```

Figure 5: Use of CBC encryption and decryption functions in main.

1.4 CFB Encryption and Decryption

CFB or Cipher Feedback is a AES mode where the block cipher turns into a stream cipher. This happens because each bytes of plain-text is XORed with a byte taken from a key-stream which results in the cipher-text. The previous cipher-text block is then encrypted and XORed with the plain-text to produce the next block. The key-stream mentioned is obtained on a byte basis, that is that the plain-text is broken into segments from anywhere between 1 byte up to the entire size of the block. Each segment is then encrypted with the block cipher the last piece of cipher-text produced so far in order to obtain the key-stream. If it is not possible, then we have the IV stream to go back to, such as in the case where in the 1st cycle there is no cipher-text yet. It's main purpose is to provide a way for encryption of individual bits or bytes which makes it possible to stream data. It also limits errors to the block that the error is contained in as well.

```python
def cfb_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    block_size = AES.block_size

    previous_block = iv

    for i in range(0, len(plaintext), block_size):
        encrypted_block = cipher.encrypt(previous_block)
        block = plaintext[i:i+block_size]
        xor_result = bytes(a ^ b for a, b in zip(block, encrypted_block))
        ciphertext += xor_result
        previous_block = xor_result

    return ciphertext

def cfb_decrypt(ciphertext, key, iv):
    return cfb_encrypt(ciphertext, key, iv)  # CFB decryption is the same as encryption
```

Figure 6: Definition of CFB Encryption and Decryption Functions

```python
# CFB Mode
iv_cfb = get_random_bytes(AES.block_size)
cfb_ciphertext = cfb_encrypt(plaintext, key, iv_cfb)
cfb_decrypted = cfb_decrypt(cfb_ciphertext, key, iv_cfb)
print(f"CFB Decrypted: {cfb_decrypted.decode()}")
```

Figure 7: Use of CFB encryption and decryption functions in main.

1.5 OFB Encryption and Decryption

OFB or Output Feedback is an AES mode where it shares a similar key-stream pattern to CFB. Each byte of plain-text is yet again XORed with a byte taken from the key-stream in order to result with the cipher-text. The difference is that the key-stream is obtained by recursively encrypting the IV value. Its purpose is to allow for pre-computation of the key-stream which makes it suitable for streaming data. Errors in this mode are also only propagated in its own block and do not affect decryption beyond the point of error.

```python
def ofb_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    block_size = AES.block_size

    previous_block = iv

    for i in range(0, len(plaintext), block_size):
        encrypted_block = cipher.encrypt(previous_block)
        block = plaintext[i:i+block_size]
        xor_result = bytes(a ^ b for a, b in zip(block, encrypted_block))
        ciphertext += xor_result
        previous_block = encrypted_block

    return ciphertext

def ofb_decrypt(ciphertext, key, iv):
    return ofb_encrypt(ciphertext, key, iv)  # OFB decryption is the same as encryption
```

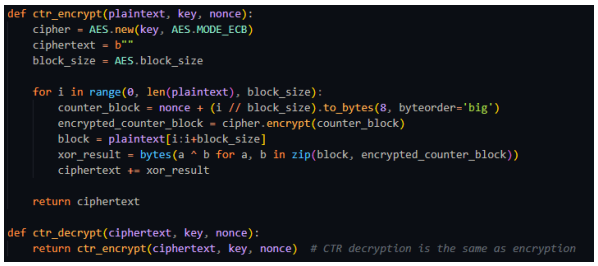
Figure 8: Definition of OFB Encryption and Decryption Functions

```python
# OFB Mode
iv_ofb = get_random_bytes(AES.block_size)
ofb_ciphertext = ofb_encrypt(plaintext, key, iv_ofb)
ofb_decrypted = ofb_decrypt(ofb_ciphertext, key, iv_ofb)
print(f"OFB Decrypted: {ofb_decrypted.decode()}")
```

Figure 9: Use of OFB encryption and decryption functions in main.

1.6 CTR Encryption and Decryption

CTR or Counter is an AES mode that turns the block cipher into a stream cipher as well. Each byte of plain-text is XOR-ed with a byte from the key-stream which results in the cipher-text. However, the key-stream is generated with a sequence of counter blocks with ECB mode. A counter block is exactly as long as the cipher block (16 bytes in this case) and consists of two separate pieces. The first piece of the counter block is a fixed nonce which is set at initialization and functions as our IV value or the value at which the counter starts. The second value of a counter block is the counter variable which gets increased by one for subsequent counter blocks and is big endian encoded. The main purpose of counter is to provide an efficient way to encrypt along with having a way to have random access and streaming applications.



```python
def ctr_encrypt(plaintext, key, nonce):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    block_size = AES.block_size

    for i in range(0, len(plaintext), block_size):
        counter_block = nonce + (i // block_size).to_bytes(8, byteorder='big')
        encrypted_counter_block = cipher.encrypt(counter_block)
        block = plaintext[i:i+block_size]
        xor_result = bytes(a ^ b for a, b in zip(block, encrypted_counter_block))
        ciphertext += xor_result

    return ciphertext

def ctr_decrypt(ciphertext, key, nonce):
    return ctr_encrypt(ciphertext, key, nonce)  # CTR decryption is the same as encryption
```

Figure 10: Definition of CTR Encryption and Decryption Functions

```python
# CTR Mode
nonce_ctr = get_random_bytes(8)
ctr_ciphertext = ctr_encrypt(plaintext, key, nonce_ctr)
ctr_decrypted = ctr_decrypt(ctr_ciphertext, key, nonce_ctr)
print(f"CTR Decrypted: {ctr_decrypted.decode()}")
```

Figure 11: Use of CTR encryption and decryption functions in main.