BIRZEIT UNIVERSITY

Faculty of Engineering & Technology
Electrical & Computer Engineering Department
APPLIED CRYPTOGRAPHY
ENCS4320

HW#2 - (Programming Assignment)

Prepared by :
Names: Abdelrhman Abed **ID:**1193191

Instructor:
Dr. Mohammed Hussein

Sec: 2

Date: 5/6/2024

Table of Contents

Table of Figure

Introduction

In this project, we implement a TEA (Tiny Encryption Algorithm) based encryption and decryption system for images using Python. TEA is a block cipher known for its simplicity and efficiency, making it suitable for applications requiring lightweight encryption. The program supports both ECB (Electronic Codebook) and CBC (Cipher Block Chaining) modes. Users can input their encryption keys, initialization vectors (for CBC mode), and specify the images to be encrypted and decrypted. This project leverages the Pillow library for image handling and the struct module for manipulating binary data.

Procedure

The procedure for encrypting and decrypting images using the TEA algorithm in ECB and CBC modes involves several detailed steps:

Initialization:

Define the TEA Class: This class encapsulates methods for encrypting and decrypting data blocks using the TEA algorithm. The class includes methods for key setup, block encryption, and decryption.
Padding and Conversion Methods: Implement methods for padding and unpadding data to ensure it fits the block size required by TEA. Also, implement methods to convert between strings and data blocks for encryption.

Image Handling:

Read Image Function: This function reads an image from a specified path, converts it into a format suitable for encryption, and handles any necessary preprocessing steps.
Save Image Function: After encryption or decryption, this function saves the processed image back to disk in the desired format.
Padding Image Data: A function to add padding to image data, ensuring it conforms to the block size requirement of the TEA algorithm.
Unpadding Image Data: After decryption, this function removes any added padding to restore the original image data.

Encryption and Decryption:

Encrypt Block: Encrypts a single block of data using TEA, applying the necessary rounds of the algorithm.
Decrypt Block: Decrypts a single block of data, reversing the TEA encryption process.
Encrypt Image (ECB Mode): Encrypts an entire image in ECB mode, processing each block of image data independently.
Decrypt Image (ECB Mode): Decrypts an image that was encrypted using ECB mode.
Encrypt Image (CBC Mode): Encrypts an image in CBC mode, where each block is XORed with the previous ciphertext block before encryption.
Decrypt Image (CBC Mode): Decrypts an image that was encrypted using CBC mode, reversing the process.

User Interaction:

Main Function: This function serves as the user interface, continuously interacting with the user. It prompts for inputs such as the encryption mode (ECB or CBC), the encryption key, the initialization vector (for CBC mode), and the image path. It validates these inputs and calls the appropriate functions for encryption or decryption based on user preferences.
Input Validation: Ensures that the user inputs are valid (e.g., key length, image path validity) before proceeding with encryption or decryption.
Looping for Continuous Interaction: The program remains active, allowing multiple encryptions and decryptions until the user decides to exit by entering 'end'.

Functions

Image Handling Functions

- **Read Image:** This function opens the specified image file, converts it to a format suitable for encryption, and prepares it for processing. It handles various image formats, ensuring compatibility and robustness.

- **Save Image:** This function saves the encrypted or decrypted image to the specified path. It ensures that the image is saved in the correct format, preserving its integrity.

- **Pad Image:** Adds necessary padding to the image data, ensuring that it fits the block size required by TEA. Padding is done in a way that can be easily removed after decryption.

- **Unpad Image:** Removes padding from the decrypted image data, restoring the original data without any extraneous padding bytes.

```python
def load_image(image_path: str) -> Tuple[bytes, str, Tuple[int, int]]:
    with Image.open(image_path) as img:
        img = img.convert('L')  # Convert to grayscale
        img_bytes = img.tobytes()
        return img_bytes, img.mode, img.size

def save_image(image_bytes: bytes, mode: str, size: Tuple[int, int], path: str):
    img = Image.frombytes(mode, size, image_bytes)
    img.save(path)

def encrypt_image(image_path: str, key: bytes, iv: bytes = None, mode: str = "ECB"):
    img_bytes, img_mode, img_size = load_image(image_path)
    tea = TEA(key)

    if mode == "ECB":
        encrypted_bytes = tea.ecb_encrypt(img_bytes)
        decrypted_bytes = tea.ecb_decrypt(encrypted_bytes)
        save_image(encrypted_bytes, img_mode, img_size, "ecb_encrypted_image.png")
        save_image(decrypted_bytes, img_mode, img_size, "ecb_decrypted_image.png")
    elif mode == "CBC":
        if iv is None:
            raise ValueError("IV must be provided for CBC mode.")
        encrypted_bytes = tea.cbc_encrypt(img_bytes, iv)
        decrypted_bytes = tea.cbc_decrypt(encrypted_bytes, iv)
        save_image(encrypted_bytes, img_mode, img_size, "cbc_encrypted_image.png")
        save_image(decrypted_bytes, img_mode, img_size, "cbc_decrypted_image.png")
```

TEA Algorithm Functions

- **Encrypt Block:** Encrypts a single 64-bit block of data using the TEA algorithm, applying multiple rounds of encryption to ensure security.

- **Decrypt Block:** Decrypts a single 64-bit block of data, reversing the encryption process to retrieve the original data.

- **Convert String to Blocks:** Converts a string of data into 64-bit blocks, suitable for encryption by the TEA algorithm.

- **Convert Blocks to String:** Converts encrypted 64-bit blocks back into a string format, reconstructing the original or encrypted data.

```python
DELTA = 0x9E3779B9

class TEA:
    def __init__(self, key: bytes):
        self.key = struct.unpack('>IIII', key)

    def encrypt_block(self, v: Tuple[int, int]) -> Tuple[int, int]:
        y, z = v
        sum = 0
        for _ in range(32):
            sum = (sum + DELTA) & 0xffffffff
            y = (y + ((z << 4) + self.key[0] ^ z + sum ^ (z >> 5) + self.key[1])) & 0xffffffff
            z = (z + ((y << 4) + self.key[2] ^ y + sum ^ (y >> 5) + self.key[3])) & 0xffffffff
        return y, z

    def decrypt_block(self, v: Tuple[int, int]) -> Tuple[int, int]:
        y, z = v
        sum = (DELTA * 32) & 0xffffffff
        for _ in range(32):
            z = (z - ((y << 4) + self.key[2] ^ y + sum ^ (y >> 5) + self.key[3])) & 0xffffffff
            y = (y - ((z << 4) + self.key[0] ^ z + sum ^ (z >> 5) + self.key[1])) & 0xffffffff
            sum = (sum - DELTA) & 0xffffffff
        return y, z
    def ecb_encrypt(self, plaintext: bytes) -> bytes:
        padded_plaintext = self.pad(plaintext)
        blocks = self.str_to_blocks(padded_plaintext)
        encrypted_blocks = [self.encrypt_block(block) for block in blocks]
        return self.blocks_to_str(encrypted_blocks)

    def ecb_decrypt(self, ciphertext: bytes) -> bytes:
        blocks = self.str_to_blocks(ciphertext)
        decrypted_blocks = [self.decrypt_block(block) for block in blocks]
        decrypted_data = self.blocks_to_str(decrypted_blocks)
        return self.unpad(decrypted_data)

    def cbc_encrypt(self, plaintext: bytes, iv: bytes) -> bytes:
        iv_block = struct.unpack('>II', iv)
        padded_plaintext = self.pad(plaintext)
        blocks = self.str_to_blocks(padded_plaintext)
        encrypted_blocks = []
        previous_block = iv_block

        for block in blocks:
            block = (block[0] ^ previous_block[0], block[1] ^ previous_block[1])
            encrypted_block = self.encrypt_block(block)
            encrypted_blocks.append(encrypted_block)
            previous_block = encrypted_block

        return self.blocks_to_str(encrypted_blocks)
```

```python
    def cbc_decrypt(self, ciphertext: bytes, iv: bytes) -> bytes:
        iv_block = struct.unpack('>II', iv)
        blocks = self.str_to_blocks(ciphertext)
        decrypted_blocks = []
        previous_block = iv_block

        for block in blocks:
            decrypted_block = self.decrypt_block(block)
            decrypted_block = (decrypted_block[0] ^ previous_block[0], decrypted_block[1] ^ previous_block[1])
            decrypted_blocks.append(decrypted_block)
            previous_block = block

        decrypted_data = self.blocks_to_str(decrypted_blocks)
        return self.unpad(decrypted_data)
```

Encrypt and Decrypt Image Functions

- **Encrypt Image (ECB Mode):** Processes the entire image in ECB mode, encrypting each block of data independently. This method highlights the potential weaknesses of ECB mode, such as patterns remaining visible in the encrypted image.

- **Decrypt Image (ECB Mode):** Decrypts an image encrypted in ECB mode, reconstructing the original image data.

- **Encrypt Image (CBC Mode):** Processes the image in CBC mode, where each block is XORed with the previous ciphertext block before encryption. This mode provides better security by ensuring that identical blocks of plaintext result in different ciphertext blocks.

- **Decrypt Image (CBC Mode):** Decrypts an image encrypted in CBC mode, reversing the process to retrieve the original image data.

```
100  def encrypt_image(image_path: str, key: bytes, iv: bytes = None, mode: str = "ECB"):
101      img_bytes, img_mode, img_size = load_image(image_path)
102      tea = TEA(key)
103
104      if mode == "ECB":
105          encrypted_bytes = tea.ecb_encrypt(img_bytes)
106          decrypted_bytes = tea.ecb_decrypt(encrypted_bytes)
107          save_image(encrypted_bytes, img_mode, img_size, "ecb_encrypted_image.png")
108          save_image(decrypted_bytes, img_mode, img_size, "ecb_decrypted_image.png")
109      elif mode == "CBC":
110          if iv is None:
111              raise ValueError("IV must be provided for CBC mode.")
112          encrypted_bytes = tea.cbc_encrypt(img_bytes, iv)
113          decrypted_bytes = tea.cbc_decrypt(encrypted_bytes, iv)
114          save_image(encrypted_bytes, img_mode, img_size, "cbc_encrypted_image.png")
115          save_image(decrypted_bytes, img_mode, img_size, "cbc_decrypted_image.png")
116
```

User Interaction Function

Main Function: This function orchestrates the overall process, interacting with the user to gather inputs, validate them, and perform the requested encryption or decryption operations. It continuously prompts the user for inputs, allowing for multiple operations until the user decides to exit.

```
117  def main():
118      while True:
119          mode = input("Enter the mode (ECB or CBC) or 'end' to exit: ").strip().upper()
120          if mode == "END":
121              break
122          if mode not in ["ECB", "CBC"]:
123              print("Invalid mode. Please enter ECB or CBC.")
124              continue
125
126          key_hex = input("Enter the key (32 hex digits): ").strip()
127          if len(key_hex) != 32:
128              print("Invalid key length. The key must be 32 hex digits.")
129              continue
130
131          try:
132              key = binascii.unhexlify(key_hex)
133          except binascii.Error:
134              print("Invalid key format. Please enter valid hex digits.")
135              continue
136
137          iv = None
138          if mode == "CBC":
139              iv_hex = input("Enter the IV (16 hex digits): ").strip()
140              if len(iv_hex) != 16:
141                  print("Invalid IV length. The IV must be 16 hex digits.")
142                  continue
143
144              try:
145                  iv = binascii.unhexlify(iv_hex)
146              except binascii.Error:
147                  print("Invalid IV format. Please enter valid hex digits.")
148                  continue
149
150          image_path = input("Enter the image path or filename with extension: ").strip()
151          try:
152              encrypt_image(image_path, key, iv, mode)
153              print(f"Encrypted and decrypted images have been saved for {mode} mode.")
154          except Exception as e:
155              print(f"An error occurred: {e}")
```

Results

After running the script, and providing the inputs, the following files are generated:

- ECB_Encryption.bmp: The image encrypted using ECB mode.
- ECB_Decryption.bmp: The decrypted image from the ECB-encrypted image.
- CBC_Encryption.bmp: The image encrypted using CBC mode.
- CBC_Decryption.bmp: The decrypted image from the CBC-encrypted image.

ECB_Encryption

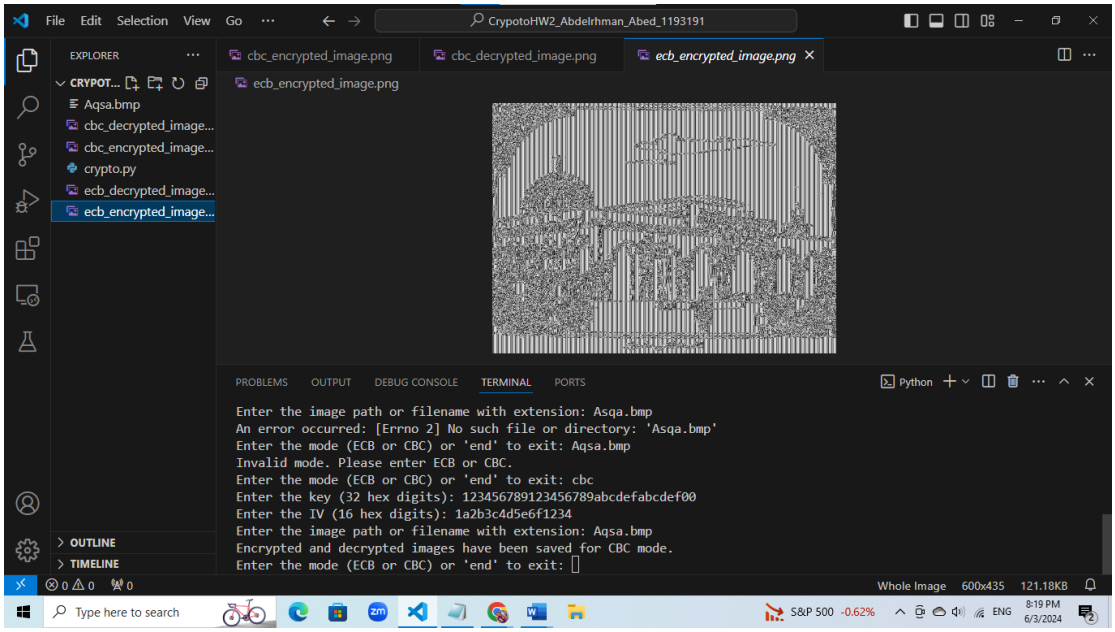

Figure 1 ECB ecnryption

ECB_Decryption

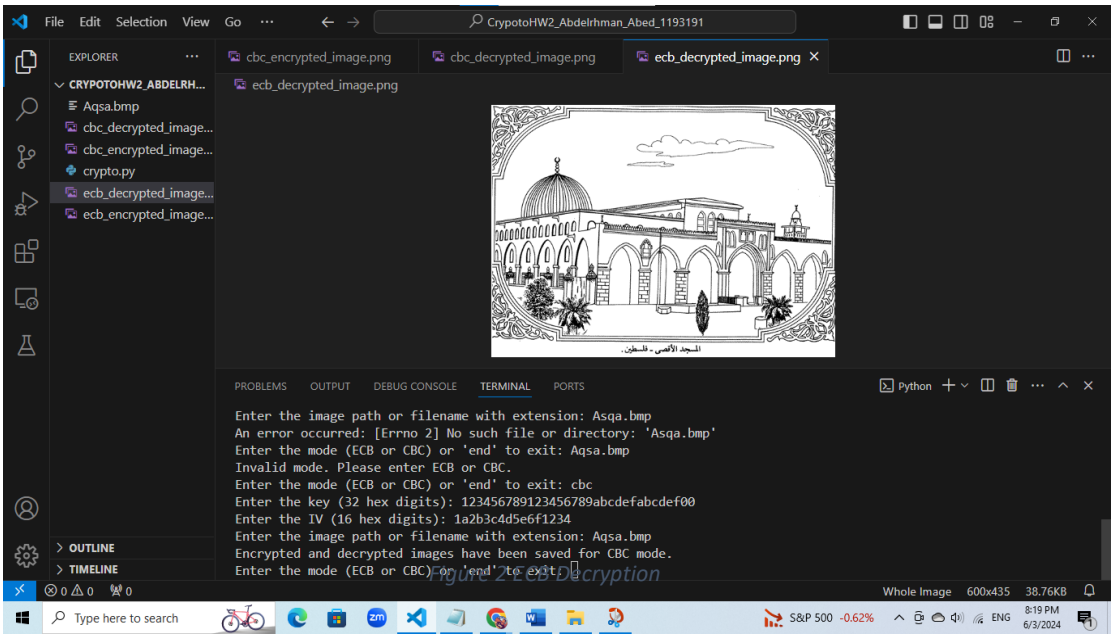

Figure 2 ECB Decryption

CBC_Encryption

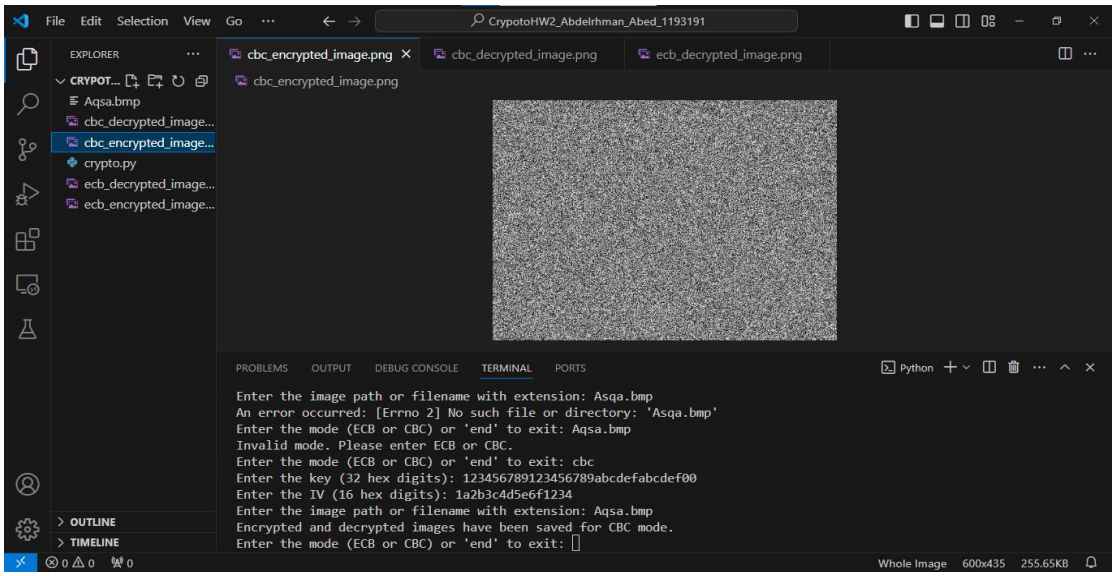

Figure 3 CBC Encryption

CBC_Decryption.bmp

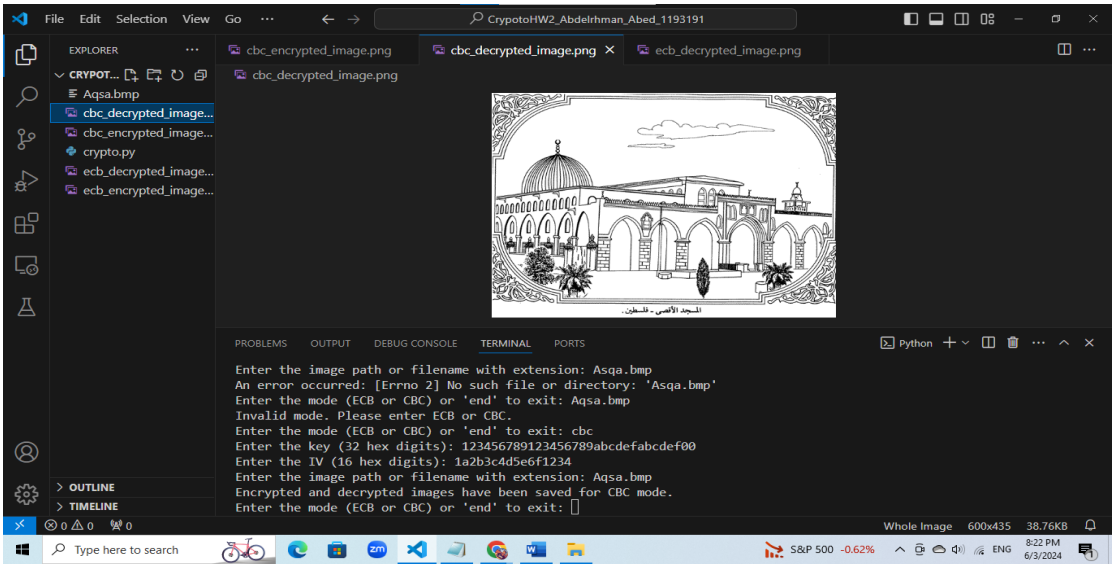

Figure 4 CBC Decryption

Conclusion

This TEA-based encryption project demonstrates how to secure images using both ECB and CBC modes. By enabling user input for keys, IVs, and image paths, the program ensures flexibility and practical usage. The program effectively encrypts and decrypts images, saving the results to disk, showcasing the versatility of TEA for lightweight encryption needs. This project not only emphasizes the importance of cryptographic techniques in securing visual data but also highlights the ease with which such techniques can be implemented and utilized in Python.