Manual Implementation of AES-ECB and AES-CBC Without Using Mode Libraries

By Shahad Alharbi

Overview of AES and Block Cipher Modes

1. AES Encryption Algorithm

The Advanced Encryption Standard (AES) is a symmetric key encryption algorithm established by the National Institute of Standards and Technology (NIST) in 2001. It is widely used for securing data due to its efficiency and robust security properties. AES operates on fixed-size blocks of data (128 bits) and supports key sizes of 128, 192, and 256 bits. The algorithm involves several rounds of processing, including substitution, permutation, mixing, and key addition.

2. Block Cipher Modes

Block cipher modes of operation define how multiple blocks of plaintext are encrypted, enabling the encryption of data larger than the block size. Two common modes are:

- Electronic Codebook (ECB): Each block is encrypted independently. Identical plaintext blocks produce the same ciphertext blocks when encrypted with the same key.

- Cipher Block Chaining (CBC): Each plaintext block is XORed with the previous ciphertext block before encryption. This introduces randomness, ensuring that identical plaintext blocks yield different ciphertext blocks.

3. Why ECB Mode is Insecure

3.1 How ECB Mode Works

In ECB mode, the plaintext is divided into blocks of equal size (e.g., 128 bits), which are then encrypted independently using the AES algorithm. While simple to implement, this leads to significant security flaws.

3.2 Vulnerabilities of ECB Mode

The primary vulnerabilities of ECB mode arise from its deterministic nature:

- Pattern Leakage: Identical plaintext blocks result in identical ciphertext blocks, revealing patterns in structured data like images or text.
- Cryptanalysis Attacks: ECB mode is susceptible to various attacks, including:
  - Known-Plaintext Attack: If an attacker knows some plaintext-ciphertext pairs, they can infer additional plaintexts.
  - Chosen-Plaintext Attack: An attacker can choose specific plaintexts to be encrypted and analyze the ciphertexts to uncover information about the encryption key or other plaintexts.

4. How CBC Mode Mitigates ECB Weaknesses

4.1 Mechanism of CBC Mode

In CBC mode, the first plaintext block is combined with an initialization vector (IV) before encryption, and each subsequent plaintext block is XORed with the previous ciphertext block. This chaining mechanism ensures that identical plaintext blocks produce different ciphertexts due to the randomness introduced by the IV.

4.2 Advantages of CBC Mode

CBC mode mitigates the vulnerabilities of ECB by:

- Breaking Pattern Repetition: Each block depends on the previous one, making identical plaintext blocks encrypt to different ciphertexts.
- Increased Security: The randomness from the IV adds an additional layer of security, making it more resistant to known-plaintext and chosen-plaintext attacks.

5. Conclusion

While ECB mode offers simplicity and ease of implementation, its deterministic nature poses significant security risks, especially with structured data. Cryptographic attacks can exploit these vulnerabilities, leading to unauthorized information disclosure. In contrast, CBC mode enhances security by chaining blocks and introducing randomness, effectively mitigating the weaknesses of ECB. Other secure modes like Galois/Counter Mode (GCM) provide additional benefits, including authentication, and should be considered for secure applications.

6. Implementation of AES Modes

6.1 Manual AES-ECB Mode Implementation

```python
# Author: Shahad Alharbi - 2210697
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import os
from PIL import Image
import base64

def encrypt_block(key, block):
    """Encrypt a single block using AES in ECB mode.

    Args:
        key (bytes): The encryption key (must be 16 bytes for AES-128).
        block (bytes): The block of data to encrypt (must be 16 bytes).

    Returns:
        bytes: The encrypted block.
    """
    aes = AES.new(key, AES.MODE_ECB)
    return aes.encrypt(block)


def decrypt_block(key, block):
    """Decrypt a single block using AES in ECB mode.

```

```python
    Args:
        key (bytes): The decryption key (must be 16 bytes for AES-128).
        block (bytes): The block of data to decrypt (must be 16 bytes).

    Returns:
        bytes: The decrypted block.
    """
    aes = AES.new(key, AES.MODE_ECB)
    return aes.decrypt(block)


def encrypt(data, key):  2 usages
    """Encrypt the data using AES in ECB mode.

    Args:
        data (bytes): The data to encrypt. It will be padded to a multiple of 16 bytes.
        key (bytes): The encryption key (must be 16 bytes long).

    Returns:
        bytes: The encrypted data.
    """
    data = pad(data, AES.block_size)  # Pad the data using PKCS#7 padding

    # Encrypt each block of data
```

```python
    encrypted = b''
    for i in range(0, len(data), 16):
        block = data[i:i + 16]  # Get the current block
        encrypted += encrypt_block(key, block)  # Encrypt the block

    return encrypted


def decrypt(encrypted_data, key):  2 usages
    """Decrypt the encrypted data using AES in ECB mode.

    Args:
        encrypted_data (bytes): The encrypted data to decrypt.
        key (bytes): The decryption key (must be 16 bytes long).

    Returns:
        bytes: The decrypted data, unpadded.
    """
    decrypted = b''
    for i in range(0, len(encrypted_data), 16):
        block = encrypted_data[i:i + 16]  # Get the current block
        decrypted += decrypt_block(key, block)  # Decrypt the block

    return unpad(decrypted, AES.block_size)  # Unpad the decrypted data
```

```python


def encrypt_image(image_path, key):
    """Encrypt the image and return the encrypted data.

    Args:
        image_path (str): The path to the image file.
        key (bytes): The encryption key (must be 16 bytes long).

    Returns:
        tuple: A tuple containing the encrypted image data (bytes) and its size (tuple).
    """
    image = Image.open(image_path)  # Open the image file
    image_data = image.tobytes()  # Get raw bytes of the image
    encrypted_data = encrypt(image_data, key)  # Encrypt the image data
    return encrypted_data, image.size  # Return the encrypted data and image size


def decrypt_image(encrypted_data, key, image_size):
    """Decrypt the image data and return a PIL Image.

    Args:
        encrypted_data (bytes): The encrypted image data.
        key (bytes): The decryption key (must be 16 bytes long).
        image_size (tuple): The size of the original image (width, height).

    Returns:
        Image: A PIL Image object of the decrypted image.
    """
    decrypted_data = decrypt(encrypted_data, key)  # Decrypt the image data
    return Image.frombytes('RGB', image_size, decrypted_data)  # Create an image from the decrypted


def encrypt_text(text, key):
    """Encrypt the text using AES in ECB mode.

    Args:
        text (str): The plaintext string to encrypt.
        key (bytes): The encryption key (must be 16 bytes long).

    Returns:
        str: The base64-encoded encrypted text.
    """
    encrypted_data = encrypt(text.encode('utf-8'), key)  # Convert text to bytes and encrypt
    return base64.b64encode(encrypted_data).decode('utf-8')  # Return base64-encoded string


def decrypt_text(encrypted_text, key):
```

```python
    """Decrypt the encrypted text using AES in ECB mode.

    Args:
        encrypted_text (str): The base64-encoded encrypted text.
        key (bytes): The decryption key (must be 16 bytes long).

    Returns:
        str: The decrypted plaintext string.
    """
    encrypted_data = base64.b64decode(encrypted_text)  # Decode the base64 string
    decrypted_data = decrypt(encrypted_data, key)  # Decrypt the data
    return decrypted_data.decode('utf-8')  # Return the decrypted string

if __name__ == '__main__':
    key = os.urandom(16)  # Generate a random 16-byte key for AES-128

    # Ask the user for the type of data to encrypt
    choice = input("Do you want to encrypt text or an image? (type 'text' or 'image'): ").strip().lower()

    if choice == 'text':
        # Get text input from the user
        text = input("Enter the text you want to encrypt: ")
```

```python
        # Encrypt the text
        encrypted_text = encrypt_text(text, key)
        print("Encrypted Text:", encrypted_text)
        # Decrypt the text
        decrypted_text = decrypt_text(encrypted_text, key)
        print("Decrypted Text:", decrypted_text)
    elif choice == 'image':
        image_path = input("Enter the path to the image you want to encrypt: ")

        # Encrypt the image
        encrypted_image_data, image_size = encrypt_image(image_path, key)

        # Decrypt the image
        decrypted_image = decrypt_image(encrypted_image_data, key, image_size)

        # Display the encrypted image as raw bytes (it won't look like a valid image)
        encrypted_image = Image.frombytes( mode: 'RGB', image_size, encrypted_image_data)
        encrypted_image.show(title="Encrypted Image (Pattern Visible)")

        # Display the decrypted image
        decrypted_image.show(title="Decrypted Image")
    else:
        print("Invalid choice. Please type 'text' or 'image'.")
```

Explanation of the Code:

- Block Functions:

  encrypt_block(key, block): Encrypts a 16-byte block.

  decrypt_block(key, block): Decrypts a 16-byte block.

- Data Functions:
  encrypt(data, key): Pads, splits, and encrypts data

decrypt(encrypted_data, key): Decrypts data and removes padding.

- Image Functions:

  encrypt_image(image_path, key): Encrypts image data and returns the encrypted bytes and size.

  decrypt_image(encrypted_data, key, image_size): Decrypts image data back to a PIL Image.

- Text Functions:

  encrypt_text(text, key): Encrypts text and returns a base64-encoded string.

  decrypt_text(encrypted_text, key): Decrypts the base64 string back to original text.

- Main Block:
  Prompts the user to choose between encrypting text or an image, generates a random key, and processes the encryption/decryption.

6.2 Manual AES-CBC Mode Implementation

```python
# Author: Shahad Alharbi - 2210697
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
from PIL import Image
import base64

BLOCK_SIZE = 16  # Block size for AES (16 bytes for AES-128)
def encrypt(data, key):
    """Encrypt the data using AES in CBC mode manually.

    Args:
        data (bytes): The data to encrypt.
        key (bytes): The encryption key (must be 16 bytes long).

    Returns:
        bytes: The encrypted data, prefixed with the IV.
    """
    iv = get_random_bytes(BLOCK_SIZE)  # Generate a random initialization vector
    cipher = AES.new(key, AES.MODE_ECB)  # Create an AES cipher in ECB mode (for manual CBC)
    padded_data = pad(data, BLOCK_SIZE)  # Pad data to be a multiple of the block size
    ciphertext = bytearray()  # Initialize ciphertext
    # Process each block of padded data
```

```python
    for i in range(0, len(padded_data), BLOCK_SIZE):
        block = padded_data[i:i + BLOCK_SIZE]

        if i == 0:
            # For the first block, XOR with the IV
            xor_block = bytes(a ^ b for a, b in zip(block, iv))
        else:
            # For subsequent blocks, XOR with the previous ciphertext block
            xor_block = bytes(a ^ b for a, b in zip(block, ciphertext[-BLOCK_SIZE:]))

        encrypted_block = cipher.encrypt(xor_block)  # Encrypt the XORed block
        ciphertext.extend(encrypted_block)  # Append the encrypted block

    return iv + ciphertext  # Prepend IV to the ciphertext

def decrypt(encrypted_data, key):  2 usages
    """Decrypt the encrypted data using AES in CBC mode manually.

    Args:
        encrypted_data (bytes): The data to decrypt, prefixed with the IV.
        key (bytes): The decryption key (must be 16 bytes long).

    Returns:
        bytes: The decrypted data, unpadded.
    """
    iv = encrypted_data[:BLOCK_SIZE]  # Extract the IV from the beginning
    cipher = AES.new(key, AES.MODE_ECB)  # Create an AES cipher in ECB mode
    decrypted_plaintext = bytearray()  # Initialize decrypted plaintext

    # Process each block of encrypted data
    for i in range(BLOCK_SIZE, len(encrypted_data), BLOCK_SIZE):
        block = encrypted_data[i:i + BLOCK_SIZE]
        decrypted_block = cipher.decrypt(block)  # Decrypt the current block

        if i == BLOCK_SIZE:
            # For the first block, XOR with the IV
            xor_block = bytes(a ^ b for a, b in zip(decrypted_block, iv))
        else:
            # For subsequent blocks, XOR with the previous ciphertext block
            xor_block = bytes(a ^ b for a, b in zip(decrypted_block, encrypted_data[i - BLOCK_SIZE:i]))

        decrypted_plaintext.extend(xor_block)  # Append the XORed plaintext

    return unpad(decrypted_plaintext, BLOCK_SIZE)  # Unpad and return the plaintext
```

```python

def encrypt_image(image_path, key):
    """Encrypt an image and return the encrypted data.

    Args:
        image_path (str): The file path of the image to encrypt.
        key (bytes): The encryption key (must be 16 bytes long).

    Returns:
        tuple: A tuple containing the encrypted image data and its size (width, height).
    """
    image = Image.open(image_path)  # Open the image file
    image_data = image.tobytes()  # Get raw bytes of the image
    encrypted_data = encrypt(image_data, key)  # Encrypt the image data
    return encrypted_data, image.size  # Return the encrypted data and image size

def decrypt_image(encrypted_data, key, image_size):
    """Decrypt the encrypted image data and return a PIL Image.

    Args:
        encrypted_data (bytes): The encrypted image data.
        key (bytes): The decryption key (must be 16 bytes long).
        image_size (tuple): The size of the original image (width, height).

    Returns:
        Image: A PIL Image object created from the decrypted data.
    """
    decrypted_data = decrypt(encrypted_data, key)  # Decrypt the image data
    return Image.frombytes('RGB', image_size, decrypted_data)  # Create an image from the decrypted

if __name__ == '__main__':
    key = get_random_bytes(16)  # Generate a random 16-byte key for AES-128

    # Prompt user for the type of data to encrypt
    choice = input("Do you want to encrypt text or an image? (type 'text' or 'image'): ").strip().lower()

    if choice == 'text':
        # Encrypt text input from user
        text = input("Enter the text you want to encrypt: ")
        encrypted_text = encrypt(text.encode('utf-8'), key)
        print("Encrypted Text (Base64):", base64.b64encode(encrypted_text).decode('utf-8'))

        # Decrypt the encrypted text and display
        decrypted_text = decrypt(encrypted_text, key).decode('utf-8')
        print("Decrypted Text:", decrypted_text)
```

```python

    elif choice == 'image':
        # Encrypt image input from user
        image_path = input("Enter the path to the image you want to encrypt: ")

        encrypted_image_data, image_size = encrypt_image(image_path, key)

        # Show the encrypted image as a distorted pattern
        encrypted_image = Image.frombytes('RGB', image_size, encrypted_image_data)
        encrypted_image.show(title="Encrypted Image (Distorted Pattern)")

        # Decrypt the image and display it
        decrypted_image = decrypt_image(encrypted_image_data, key, image_size)
        decrypted_image.show(title="Decrypted Image")
    else:
        print("Invalid choice. Please type 'text' or 'image'.")
```

Explanation of the Code:

- Constants
  BLOCK_SIZE: Defines the block size for AES (16 bytes).

- Encryption Function
  encrypt(data, key): Encrypts data using AES-CBC. It generates a random initialization vector (IV), pads the data, and XORs each block with the IV or the previous ciphertext block.

- Decryption Function

  decrypt(encrypted_data, key): Decrypts data by extracting the IV, decrypting each block, and XORing with the IV or the previous ciphertext.

- Image Functions

  encrypt_image(image_path, key): Encrypts the raw byte data of an image and returns the encrypted data and size.

  decrypt_image(encrypted_data, key, image_size): Decrypts the encrypted image data and reconstructs it into a PIL Image.

- Main Execution Block
  Prompts the user to choose between encrypting text or an image, generates a random key, and displays the encrypted and decrypted results.

7. Step 3: Perform Cryptanalysis on AES-ECB

7.1 Overview

This step analyzes the ciphertext produced by the AES-ECB encryption process to identify repeated blocks, demonstrating a significant vulnerability in ECB mode.

7.2 Code Implementation

The following Python function detects repeated blocks in the encrypted data:

```python
def detect_repeated_blocks(encrypted_text):  1 usage
    """Detect repeated blocks in the encrypted data."""
    encrypted_data = base64.b64decode(encrypted_text)
    blocks = [encrypted_data[i:i + 16] for i in range(0, len(encrypted_data), 16)]
    seen = {}
    repeated_blocks = set()

    for block in blocks:
        if block in seen:
            repeated_blocks.add(block)
        else:
            seen[block] = True

    return repeated_blocks
```

7.3 Output Analysis

When the program is executed with the text input "hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello", the following results are observed:

```
C:\Users\shdal\PycharmProjects\pythonProject5\.venv\Scripts\python.exe C:\Users\shdal\PycharmProjects\pythonProject5\CBCimage.py
Do you want to encrypt text or an image? (type 'text' or 'image'): text
Enter the text you want to encrypt: hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohel
Encrypted Text: GT7ZFYxZZKRjvZeSebjTrSxsOGzcneH6HPQrUykshGcfMXSXitYnhEvFV5YRkqgauUvbpnoekf8hJTnmZnCoW8vuJXNyJQqHbvJyTdkpmUIZPtkVjFlkpGO9l5J5uNOt
Repeated Blocks Detected: 2
Decrypted Text: hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello

Process finished with exit code 0
```

The detection of repeated blocks indicates that two identical blocks were found in the encrypted data. This is a significant observation, as it highlights one of the vulnerabilities of the AES-ECB mode, where identical plaintext blocks produce identical ciphertext blocks.

Summary

The detection of repeated blocks shows the weaknesses of ECB mode, where identical plaintext blocks result in identical ciphertext. This reinforces the need for more secure encryption methods, like CBC, which can prevent such vulnerabilities.

8. Comparison of AES-ECB and AES-CBC

8.1 AES-ECB

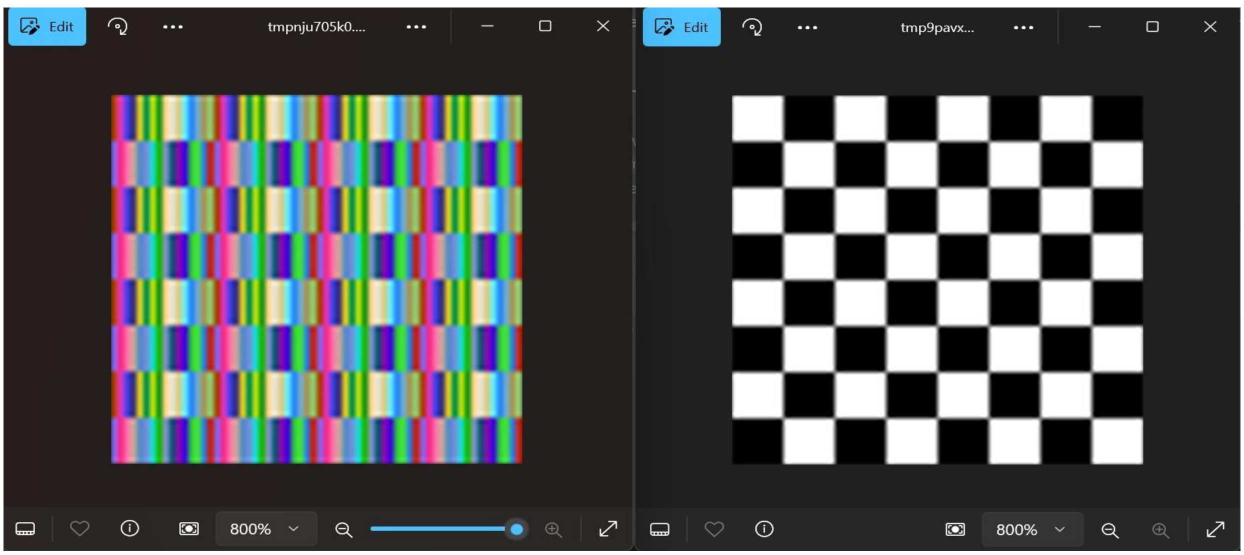
Text Analysis:

```
Do you want to encrypt text or an image? (type 'text' or 'image'): text
Enter the text you want to encrypt: hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello
Encrypted Text: NE+arNVsxRBfWHjZLcCRCRuSXYFr/Zu4w7PQMsZT3ZoaNnQgPtrVZ7D6SN7T3H5X8mwaFD48S3PYwjWNy52q2wnek75K2y/l1WBbJlE/LIPtTf2t7wqggZ74HXSH9nA
Decrypted Text: hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello

Process finished with exit code 0
```

- Deterministic Output: Identical plaintext blocks yield identical ciphertext blocks, exposing structural patterns.
- Pattern Vulnerability: Repeated ciphertext blocks make ECB highly vulnerable to pattern analysis, risking unauthorized access to sensitive information.
- Security Implications: Due to its significant security risks, ECB mode is unsuitable for sensitive data.

Image Analysis:

```
C:\Users\shdal\PycharmProjects\pythonProject5\.venv\Scripts\python.exe C:\Users\shdal\PycharmProjects\pythonProject5\ebc4.py
Do you want to encrypt text or an image? (type 'text' or 'image'): image
Enter the path to the image you want to encrypt: checkerboard_64x64.png

Process finished with exit code 0
```



- **Deterministic Output**: Identical pixel data in an image produces identical ciphertext blocks, revealing visible patterns.
- **Pattern Vulnerability**: Recognizable patterns in encrypted images can be exploited, risking exposure of sensitive visual data.
- **Security Implications**: The presence of patterns necessitates using more secure modes like CBC or GCM to protect against vulnerabilitie

8.2 AES-CBC

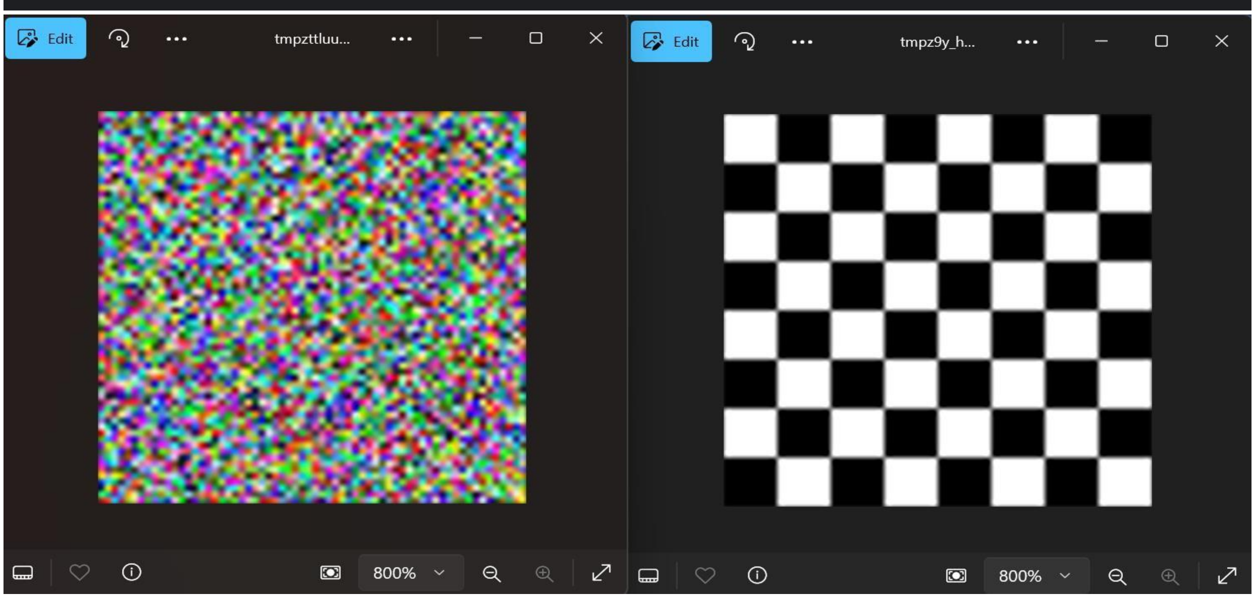
Text Analysis:

```
Do you want to encrypt text or an image? (type 'text' or 'image'): text
Enter the text you want to encrypt: hellohellohellohellohellohellohellohello
Encrypted Text (Base64): Z0MoKfeG32pLJCm+nDtMYv+TRHLEZZsA7nQDhxlVNfvOJW+/ZwnufpVhIL//9MVEI2dW8+puR8TIKlyxyhZW7A==
Decrypted Text: hellohellohellohellohellohellohellohello

Process finished with exit code 0
```

- **Randomness**: The use of an IV ensures that even identical plaintext blocks result in different ciphertext blocks, making it harder to discern patterns.
- **Enhanced Security**: CBC mode reduces the risk of pattern analysis, making it more suitable for encrypting sensitive text data.

Image Analysis:

```
C:\Users\shdal\PycharmProjects\pythonProject5\.venv\Scripts\python.exe C:\Users
Do you want to encrypt text or an image? (type 'text' or 'image'): image
Enter the path to the image you want to encrypt: checkerboard_64x64.png

Process finished with exit code 0
```



- **Random IV Usage:** Each encryption session utilizes a unique IV, preventing identical pixel data from producing identical ciphertext.
- **Pattern Concealment**: CBC mode effectively hides patterns in images, enhancing security

against visual data analysis.

- **Security Advantages:** The chaining mechanism of CBC ensures that each block influences the next, further obscuring patterns in the encrypted image.

Summary

In both text and image encryption, AES-ECB exhibits significant vulnerabilities due to its deterministic nature, leading to recognizable patterns in ciphertext. Conversely, AES-CBC enhances security by introducing randomness through IVs and chaining blocks, making it a better choice for protecting sensitive data. This analysis emphasizes the importance of selecting appropriate encryption methods based on the type of data being secured to ensure confidentiality and security.