Assignment 1: AES
ELL895 - Network Security

Animesh Lohar - 2024EET2368

May 6, 2025

1 Introduction

This report provides an analysis of a Python implementation of the Advanced Encryption Standard (AES) algorithm, specifically designed for encrypting and decrypting image files. The implementation is divided into two Python scripts: `aes_encryption.py` and `aes_decryption.py`. The AES algorithm is a symmetric block cipher widely used for securing digital data. It operates on fixed-size blocks of plaintext (128 bits, or 16 bytes) and transforms them into ciphertext blocks of the same size using a secret key.

The implementation utilizes standard Python libraries such as `PIL` (Pillow) for image handling, `numpy` for array manipulation, and `hashlib` for key derivation.

2 AES Algorithm Overview

AES is an iterative block cipher that performs a series of transformations on the plaintext block in multiple rounds. The number of rounds depends on the key size: 10 rounds for a 128-bit key, 12 rounds for a 192-bit key, and 14 rounds for a 256-bit key. Each round, except the last, consists of four main transformations:

1. **SubBytes:** Each byte in the state is replaced with another byte using a substitution box (S-box). This provides non-linearity.
2. **ShiftRows:** The rows of the state are cyclically shifted by different offsets. This provides diffusion across columns.
3. **MixColumns:** Each column of the state is transformed by multiplying it with a fixed matrix in the Galois field GF(2^8). This provides diffusion across rows. (The last round omits this step).
4. **AddRoundKey:** The state is XORed with the round key, which is derived from the original secret key through a key expansion process.

The decryption process involves reversing these transformations in reverse order, using inverse operations for each step.

3 Code Analysis: `aes_encryption.py`

The `aes_encryption.py` script implements the encryption logic. The core functionality is encapsulated within the `AESEncryptor` class.

3.1 Class Structure and Initialization

The `AESEncryptor` class is initialized with a user-provided key. This key is then processed:

- **Key Padding/Derivation:** The `_pad_key` method uses SHA-256 to hash the input key and then truncates or pads the hash to match the required key size (16, 24, or 32 bytes) based on the length of the input key. This provides a fixed-size key suitable for AES.
- **Key Expansion:** The `_key_expansion` method generates the round keys from the padded key. This process involves rotating words, substituting bytes using the S-box, and XORing with round constants (`RCON`). The number of rounds (`nr`) is determined based on the key size (`nk`).

3.2 Core AES Transformations (Encryption)

The class implements the standard AES encryption transformations:

- `_sub_bytes`: Performs the SubBytes transformation using the defined `SBOX`.
- `_shift_rows`: Performs the cyclic shift of rows.
- `_mix_columns`: Performs the MixColumns transformation. This involves multiplication in the Galois field GF(2^8), which is handled by the `_gf_mul` method.
- `_add_round_key`: Performs the XOR operation with the round key.

3.3 Block Encryption

The `_encrypt_block` method orchestrates the AES rounds for a single 16-byte block of data. It applies the initial AddRoundKey, followed by the main rounds (SubBytes, ShiftRows, MixColumns, AddRoundKey), and finally the last round (SubBytes, ShiftRows, AddRoundKey).

3.4 Image Encryption

The `encrypt_image` method handles the encryption of an entire image:

- It opens the image using PIL.
- Converts the image data to a byte sequence.
- Applies PKCS7 padding using the `_pad_data` method to ensure the data length is a multiple of 16 bytes.
- Iterates through the padded data, encrypting each 16-byte block using `_encrypt_block`.
- Reconstructs an image from the encrypted data using the original image's mode and size.
- Saves the encrypted image.

3.5 Padding

The `_pad_data` method implements PKCS7 padding. It calculates the number of bytes needed to reach a multiple of 16 and appends that many bytes, each containing the padding length.

```python
from PIL import Image
import numpy as np
import hashlib

class AESEncryptor:
    SBOX = [...] # S-box definition

    RCON = [...] # Round constants definition

    def __init__(self, key):
        self.key = self._pad_key(key)
        self.nk = len(self.key) // 4
        self.nr = {4:10, 6:12, 8:14}[self.nk]
        self.round_keys = self._key_expansion()

    def _pad_key(self, key):
        hash_obj = hashlib.sha256(key.encode()).digest()
        lengths = {16:16, 24:24, 32:32}
        return hash_obj[:lengths.get(len(key), 32)]

    def _key_expansion(self):
        # ... key expansion logic ...
        pass

    def _sub_bytes(self, state):
        return [[self.SBOX[b] for b in row] for row in state]

    def _shift_rows(self, state):
        # ... shift rows logic ...
        pass

    def _mix_columns(self, state):
        # ... mix columns logic ...
        pass

    def _gf_mul(self, a, b):
        # ... GF(2^8) multiplication logic ...
        pass

    def _add_round_key(self, state, round_key):
        return [[b ^ k for b, k in zip(row, key)]
                for row, key in zip(state, round_key)]

    def _encrypt_block(self, block):
        state = self._bytes_to_state(block)
        state = self._add_round_key(state, self.round_keys[0])
        for round in range(1, self.nr):
            state = self._sub_bytes(state)
            state = self._shift_rows(state)
            state = self._mix_columns(state)
            state = self._add_round_key(state, self.round_keys[round])
        state = self._sub_bytes(state)
        state = self._shift_rows(state)
        state = self._add_round_key(state, self.round_keys[self.nr])
        return self._state_to_bytes(state)

    def encrypt_image(self, image_path, output_path):
        # ... image encryption logic ...
        pass

    def _pad_data(self, data):
        """Add PKCS#7 padding"""
        pad_len = 16 - (len(data) % 16)
        return data + bytes([pad_len] * pad_len)

# ... __main__ block ...
```

Listing 1: aes_encryption.py (Excerpts)

4 Code Analysis: `aes_decryption.py`

The `aes_decryption.py` script implements the decryption logic, largely mirroring the encryption process with inverse operations. It defines the `AESDecryptor` class.

4.1 Class Structure and Initialization

The initialization is similar to the encryptor, using the same key padding and key expansion logic. It also imports the `AESEncryptor` to reuse the SBOX and RCON constants. A crucial addition is the `INVERSE_SBOX`, which is the inverse mapping of the SBOX.

4.2 Core AES Transformations (Decryption)

The class implements the inverse transformations:

- `_inverse_sub_bytes`: Performs the inverse SubBytes transformation using the `INVERSE_SBOX`. Includes a basic check for valid byte values.
- `_inverse_shift_rows`: Performs the inverse cyclic shift of rows.
- `_inverse_mix_columns`: Performs the inverse MixColumns transformation using different fixed coefficients for multiplication in GF(2^8).
- `_add_round_key`: The AddRoundKey operation is its own inverse, so the same method is used.

4.3 Block Decryption

The _decrypt_block method reverses the encryption process for a 16-byte block. It applies the final AddRoundKey, followed by the inverse main rounds (InverseMixColumns, AddRoundKey, InverseSubBytes, InverseShiftRows), and finally the initial round (AddRoundKey, InverseSubBytes, InverseShiftRows).

4.4 Image Decryption

The decrypt_image method handles image decryption:

- Opens the encrypted image.
- Converts the image data to a byte sequence.
- Iterates through the data, decrypting each 16-byte block using _decrypt_block. A zero-padding is added if a block is shorter than 16 bytes, although this might indicate an issue with the encrypted data format if it's not a multiple of 16.
- Unpads the decrypted data using the _unpad_data method.
- Reconstructs an image from the unpadded data.
- Saves the decrypted image.

4.5 Unpadding

The _unpad_data method removes the PKCS7 padding. It reads the last byte to determine the padding length and truncates the data accordingly. It includes a basic check to prevent removing more data than the block size.

```python
from PIL import Image
import numpy as np
import hashlib

from aes_encryption import AESEncryptor

class AESDecryptor:
    SBOX = AESEncryptor.SBOX
    INVERSE_SBOX = [...] # Inverse S-box definition

    RCON = AESEncryptor.RCON

    def __init__(self, key):
        self.key = self._pad_key(key)
        self.nk = len(self.key) // 4
        self.nr = {4:10, 6:12, 8:14}[self.nk]
        self.round_keys = self._key_expansion()

    def _pad_key(self, key):
        # ... key padding logic (same as encryptor) ...
        pass

    def _key_expansion(self):
        # ... key expansion logic (same as encryptor) ...
        pass

    def _inverse_sub_bytes(self, state):
        # ... inverse sub bytes logic with check ...
        pass

    def _inverse_shift_rows(self, state):
        # ... inverse shift rows logic ...
        pass

    def _inverse_mix_columns(self, state):
        # ... inverse mix columns logic ...
        pass

    def _gf_mul(self, a, b):
        # ... GF(2^8) multiplication logic (same as encryptor) ...
        pass

    def _add_round_key(self, state, round_key):
        # ... add round key logic (same as encryptor) ...
        pass

    def _decrypt_block(self, block):
        # ... block decryption logic ...
        pass

    def _unpad_data(self, data):
        pad_len = data[-1]
        return data[:-pad_len] if pad_len <= 16 else data

    def decrypt_image(self, image_path, output_path):
        # ... image decryption logic ...
        pass

# ... __main__ block ...
```

Listing 2: aes_decryption.py (Excerpts)

5 Observations and Potential Enhancements

5.1 Key Derivation

Using SHA-256 to derive the key is a common and reasonable approach. It ensures that keys of varying lengths are transformed into a fixed-size key suitable for AES.

5.2 Image Handling

The approach of converting the image to bytes, encrypting/decrypting the bytes, and then converting back to an image is straightforward. However, this treats the image data as a flat sequence of bytes without considering the image's structure (width, height, color channels). While this works for simple image formats and modes, it might not be ideal for all cases and could potentially impact image integrity if not handled carefully.

5.3 Padding

PKCS7 padding is a standard and correct method for ensuring block alignment. The unpadding logic correctly removes the padding based on the last byte.

5.4 Galois Field Multiplication

The `_gf_mul` function correctly implements multiplication in GF(2^8) for the MixColumns and InverseMixColumns transformations.

5.5 Potential Enhancements

- **Error Handling:** More robust error handling could be added, especially for file operations and invalid input data.
- **Different AES Modes:** The current implementation appears to be a basic Electronic Codebook (ECB) mode implementation, as each block is encrypted independently. ECB is generally not recommended for images or data with patterns, as it can reveal those patterns in the ciphertext. Implementing other modes like CBC (Cipher Block Chaining), OFB (Output Feedback), or CTR (Counter) would provide better security. This would require managing an Initialization Vector (IV).
- **Performance:** For very large images, performance could be a consideration. Using optimized libraries or considering block processing in chunks might be beneficial.

- **Key Management:** In a real-world application, secure key management is crucial. This implementation assumes the key is readily available to both the encryptor and decryptor.
- **Image Mode Handling:** The code assumes the image mode and size are preserved. This might not always be the case depending on the image format and how the encrypted bytes are interpreted.

6 Conclusion

The provided Python code implements a functional AES encryption and decryption system for images. It correctly implements the core AES transformations and handles image data conversion and padding. While the implementation is a good starting point, particularly for understanding the AES algorithm, using a more secure mode of operation (like CBC or CTR) and incorporating more robust error handling would improve its practical applicability and security.