**Python AES-128: A Simple Implementation**

<https://github.com/ash0545/simple-aes/>

**REPORT**

**Ashwin Santhosh**

**CS22B1005**

**Implementation**

The AES-128 implementation follows the standard encryption and decryption procedures defined by the AES specification (FIPS-197). The key expansion, substitution, permutation, and XOR operations are implemented as separate functions for clarity and modularity.

The key steps for the implementation include:

* Key Expansion using RotWord, SubWord, and Rcon
* SubBytes operation using S-box lookup
* ShiftRows transformation to rearrange bytes
* MixColumns transformation for diffusion
* AddRoundKey operation for XOR with round keys
* Final round omits MixColumns for correctness

**Code**

from lookups\_constants import (

    sbox,

    inv\_sbox,

    rcon,

    multiply\_by\_2,

    multiply\_by\_3,

    multiply\_by\_9,

    multiply\_by\_11,

    multiply\_by\_13,

    multiply\_by\_14,

)

*# ==============================================================================*

*#                            KEY EXPANSION*

*# ==============================================================================*

*#  AES-128 => we need 11 keys*

*# "Word" as in 4 bytes*

def RotWord(word):

    return word[1:] + [word[0]]

def SubWord(word):

    return [sbox[x] for x in word]

def Rcon(x):

    return [rcon[x], 0x00, 0x00, 0x00]

*# Assuming key as a 2d array for ease of implementation*

def KeyExpansion(key):

    expanded\_keys = [key]

    for round in range(10):

        prev\_round\_key = expanded\_keys[-1]

        curr\_round\_key = []

*# Extract last column*

        last\_col = [prev\_round\_key[row][-1] for row in range(4)]

*# Perform transformations*

        rotated = RotWord(last\_col)

        substituted = SubWord(rotated)

        rcon\_col = Rcon(round + 1)  *# Rcon index should start from 1*

*# Compute first column of new round key*

        first\_col = [

            substituted[j] ^ prev\_round\_key[j][0] ^ rcon\_col[j] for j in range(4)

        ]

        curr\_round\_key.append(first\_col)

*# Compute remaining columns*

        for col in range(1, 4):

            new\_col = [

                curr\_round\_key[col - 1][j] ^ prev\_round\_key[j][col] for j in range(4)

            ]

            curr\_round\_key.append(new\_col)

*# Transpose to maintain column-major order*

        curr\_round\_key = [

            [curr\_round\_key[col][row] for col in range(4)] for row in range(4)

        ]

        expanded\_keys.append(curr\_round\_key)

    return expanded\_keys

*# ==============================================================================*

*#                            AES ENCRYPTION FUNCTIONS*

*# ==============================================================================*

def SubBytes(state):

    return [[sbox[elem] for elem in row] for row in state]

def ShiftRows(state):

*# Row 1: No shift*

    state[1] = state[1][1:] + state[1][:1]

    state[2] = state[2][2:] + state[2][:2]

    state[3] = state[3][3:] + state[3][:3]

    return state

def MixColumns(state):

*# Column i : state[0][i], state[1][i], state[2][i], state[3][i]*

    for i in range(4):

        a0, a1, a2, a3 = state[0][i], state[1][i], state[2][i], state[3][i]

        state[0][i] = multiply\_by\_2[a0] ^ multiply\_by\_3[a1] ^ a2 ^ a3

        state[1][i] = multiply\_by\_2[a1] ^ multiply\_by\_3[a2] ^ a3 ^ a0

        state[2][i] = multiply\_by\_2[a2] ^ multiply\_by\_3[a3] ^ a0 ^ a1

        state[3][i] = multiply\_by\_2[a3] ^ multiply\_by\_3[a0] ^ a1 ^ a2

    return state

def AddRoundKey(state, round\_key):

    for i in range(4):

        for j in range(4):

            state[i][j] ^= round\_key[i][j]

    return state

def encrypt(state, round\_keys):

*# pre-whitening ie., xor-ing state with first round key*

    state = AddRoundKey(state, round\_keys[0])

    print(f"Pre-whitening: {matrix\_to\_hex\_string(state)}")

*# n - 1 rounds*

    for i in range(1, 10):

        print(f"Round {i} start: {matrix\_to\_hex\_string(state)}")

        state = SubBytes(state)

        print(f"SubBytes: {matrix\_to\_hex\_string(state)}")

        state = ShiftRows(state)

        print(f"ShiftRows: {matrix\_to\_hex\_string(state)}")

        state = MixColumns(state)

        print(f"MixColumns: {matrix\_to\_hex\_string(state)}")

        state = AddRoundKey(state, round\_keys[i])

        print()

*# last round with absence of MixColumns*

    print(f"Last Round start: {matrix\_to\_hex\_string(state)}")

    state = SubBytes(state)

    print(f"SubBytes: {matrix\_to\_hex\_string(state)}")

    state = ShiftRows(state)

    print(f"ShiftRows: {matrix\_to\_hex\_string(state)}")

    state = AddRoundKey(state, round\_keys[10])

    print()

    return state

*# ==============================================================================*

*#                            AES DECRYPTION FUNCTIONS*

*# ==============================================================================*

def InvShiftRows(state):

*# Row 1: No shift*

    state[1] = state[1][3:] + state[1][:3]

    state[2] = state[2][2:] + state[2][:2]

    state[3] = state[3][1:] + state[3][:1]

    return state

def InvSubBytes(state):

    return [[inv\_sbox[elem] for elem in row] for row in state]

def InvMixColumns(state):

*# Column i : state[0][i], state[1][i], state[2][i], state[3][i]*

    for i in range(4):

        a0, a1, a2, a3 = state[0][i], state[1][i], state[2][i], state[3][i]

        state[0][i] = (

            multiply\_by\_14[a0]

            ^ multiply\_by\_11[a1]

            ^ multiply\_by\_13[a2]

            ^ multiply\_by\_9[a3]

        )

        state[1][i] = (

            multiply\_by\_14[a1]

            ^ multiply\_by\_11[a2]

            ^ multiply\_by\_13[a3]

            ^ multiply\_by\_9[a0]

        )

        state[2][i] = (

            multiply\_by\_14[a2]

            ^ multiply\_by\_11[a3]

            ^ multiply\_by\_13[a0]

            ^ multiply\_by\_9[a1]

        )

        state[3][i] = (

            multiply\_by\_14[a3]

            ^ multiply\_by\_11[a0]

            ^ multiply\_by\_13[a1]

            ^ multiply\_by\_9[a2]

        )

    return state

def decrypt(state, round\_keys):

*# inverse of pre-whitening ie., xor-ing state with last round key*

    state = AddRoundKey(state, round\_keys[10])

    print(f"Inverse Pre-whitening: {matrix\_to\_hex\_string(state)}")

*# n - 1 rounds*

    for i in range(9, 0, -1):

        print(f"Inverse Round {i} start: {matrix\_to\_hex\_string(state)}")

        state = InvShiftRows(state)

        print(f"InvShiftRows: {matrix\_to\_hex\_string(state)}")

        state = InvSubBytes(state)

        print(f"InvSubBytes: {matrix\_to\_hex\_string(state)}")

        state = AddRoundKey(state, round\_keys[i])

        print(f"AddRoundKey: {matrix\_to\_hex\_string(state)}")

        state = InvMixColumns(state)

        print()

*# last round with absence of MixColumns*

    print(f"Last Round start: {matrix\_to\_hex\_string(state)}")

    state = InvShiftRows(state)

    print(f"InvShiftRows: {matrix\_to\_hex\_string(state)}")

    state = InvSubBytes(state)

    print(f"InvSubBytes: {matrix\_to\_hex\_string(state)}")

    state = AddRoundKey(state, round\_keys[0])

    print()

    return state

*# ==============================================================================*

*#                           HELPER FUNCTIONS*

*# ==============================================================================*

def hex\_string\_to\_matrix(hex\_string):

    """Converts a 32-character hex string into a 4x4 column-major order matrix."""

    bytes\_list = [int(hex\_string[i : i + 2], 16) for i in range(0, len(hex\_string), 2)]

    return [bytes\_list[i::4] for i in range(4)]  *# Convert to 4x4 matrix*

def matrix\_to\_hex\_string(matrix):

    """Converts a 4x4 matrix back to a hex string."""

    flat\_list = [

        matrix[row][col] for col in range(4) for row in range(4)

    ]  *# Column-major order*

    return "".join(f"{byte:02x}" for byte in flat\_list)

*# ==============================================================================*

*#                                   MAIN*

*# ==============================================================================*

if \_\_name\_\_ == "\_\_main\_\_":

*# Example key as a hex string (Taken from example vector in Appendix C of AES Standard)*

    hex\_key = "000102030405060708090a0b0c0d0e0f"

*# hex\_key = "d6aa74fdd2af72fadaa678f1d6ab76fe"*

    print(f"Initial Key: {hex\_key}")

    key\_matrix = hex\_string\_to\_matrix(hex\_key)

*# Generate round keys*

    round\_keys = KeyExpansion(key\_matrix)

*# Print the round keys in hex format*

    print("Expanded Round Keys:")

    for i, round\_key in enumerate(round\_keys):

        print(f"Round {i}: {matrix\_to\_hex\_string(round\_key)}")

*# Example plaintext as hex string (Taken from example vector in Appendix C of AES Standard)*

    hex\_plain\_text = "00112233445566778899aabbccddeeff"

    start\_state = hex\_string\_to\_matrix(hex\_plain\_text)

    print("Plaintext Start State:")

    for row in start\_state:

        print([hex(x)[2:].zfill(2) for x in row])  *# Print matrix in hex format*

    print()

    encrypted = encrypt(start\_state, round\_keys)

    print(f"Encryption Result: {matrix\_to\_hex\_string(encrypted)}")

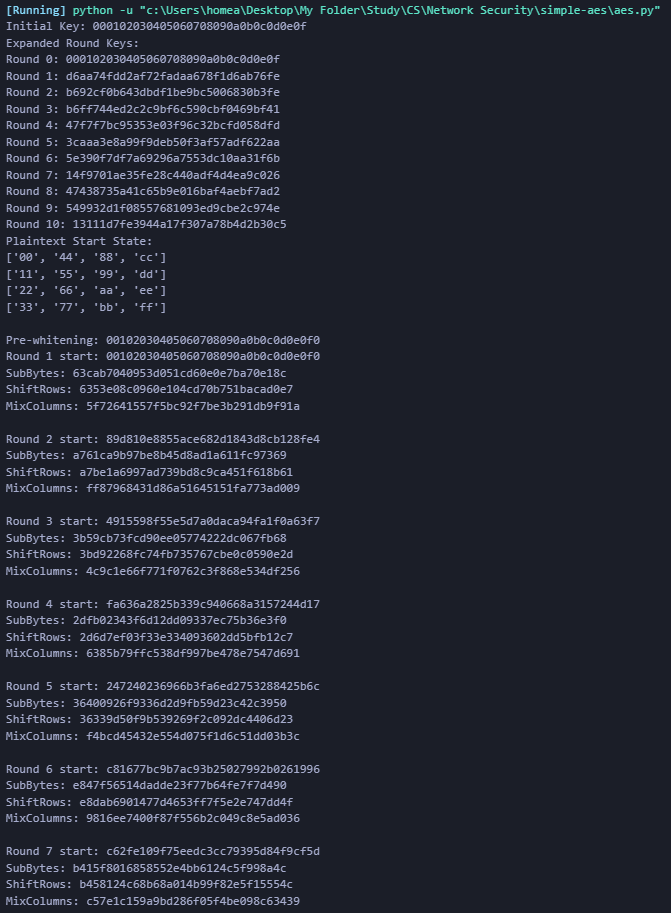
    print()

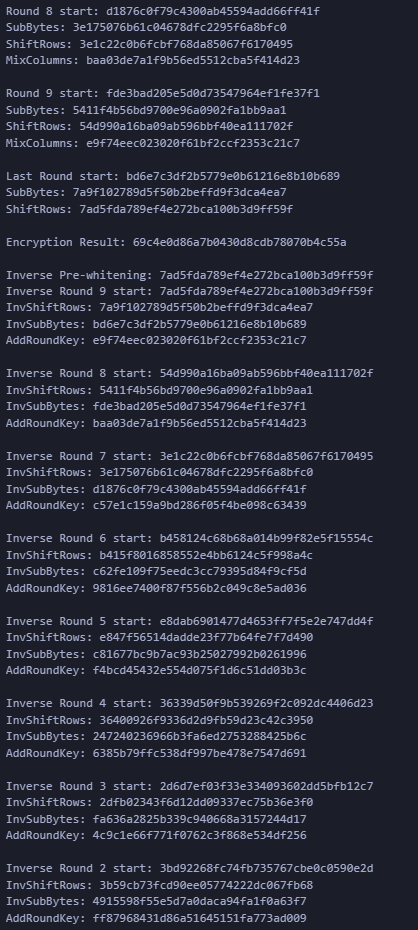
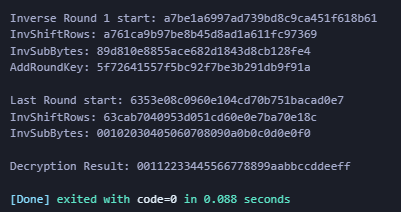
    decrypted = decrypt(encrypted, round\_keys)

    print(f"Decryption Result: {matrix\_to\_hex\_string(decrypted)}")

**Results**

The implementation was tested using standard AES test vectors. The encryption and decryption results matched the expected values, verifying the correctness of the implementation.





**References**

1. David Wong’s Block Breakers: <https://davidwong.fr/blockbreakers/aes.html>
2. The official AES standard: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
3. boppreh’s AES Implementation in Python: <https://github.com/boppreh/aes>