BRACT's
Vishwakarma Institute of Information Technology
Department of Computer Engineering(Software Engineering)

Submitted By :

Roll No.	Name	PRN No.
3101070	Pranita Narayan Kute	22420110

ASSIGNMENT NO. 4

1. **Title** - Implement Symmetric Key Encryption using AES
2. **Aim -** To implement symmetric key encryption using the Advanced Encryption Standard (AES) algorithm.
3. **Objective**

   a. To study the working of symmetric key cryptography.

   b. To understand the structure and functioning of the AES algorithm (SubBytes, ShiftRows, MixColumns, AddRoundKey).

   c. To implement encryption and decryption using AES-128 in CTR mode.

   d. To demonstrate secure data transmission using a shared secret key and random nonce.

   e. To analyze the strengths and advantages of AES over older standards such as DES.

4. **Pre-requisite Knowledge**

   a. Basics of cryptography – plain text, cipher text, encryption, decryption.

   b. Difference between symmetric and asymmetric key encryption.

   c. Understanding of block ciphers, key sizes (128/192/256 bits), and block modes (CTR, CBC, etc.).

   d. Knowledge of the AES algorithm structure (key expansion, 10/12/14 rounds depending on key size).

   e. Familiarity with a programming language (e.g., Python/Java/C++) to implement AES.

   f. Basic understanding of cryptographic libraries/tools or finite field arithmetic.

5. Theory

- **Introduction:**

1. Advanced Encryption Standard (AES) is a highly trusted encryption algorithm used to secure data by converting it into an unreadable format without the proper key.
2. It was developed by the National Institute of Standards and Technology (NIST) in 2001.
3. It is widely used today as it is much stronger than DES and triple DES despite being harder to implement.
4. AES encryption uses various key lengths (128, 192, or 256 bits) to provide strong protection against unauthorized access.
5. This data security measure is efficient and widely implemented in securing internet communication, protecting sensitive data, and encrypting files.
6. AES, a cornerstone of modern cryptography, is recognized globally for its ability to keep information safe from cyber threats.

- AES is a Block Cipher.
- The key size can be 128/192/256 bits.
- Encrypts data in blocks of 128 bits each.

That means it takes 128 bits as input and outputs 128 bits of encrypted cipher text. AES relies on the substitution-permutation network principle, which is performed using a series of linked operations that involve replacing and shuffling the input data.

6. Code

a. Custom S-box: Every byte in the standard AES S-box is XORed with 0x11 as a demonstration of modifying S-box values.

b. AES Core Steps: Key expansion, SubBytes (uses custom S-box), ShiftRows, MixColumns, AddRoundKey.

c. Output: Prints the encrypted 16-byte ciphertext as hexadecimal.

Code

```
from typing import List


# -------------------- Modified AES S-box --------------------

# Only the first few entries have been slightly changed as an example

Sbox = [
```

0x65, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,

0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,

0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,

0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,

0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,

0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,

0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,

0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,

0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,

0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,

0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,

0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,

0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,

0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,

0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,

```
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54,
0xbb, 0x16

]



# Round constants used during key expansion

Rcon = [0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1B,0x36]



# -------------------- AES Helper Functions --------------------


# Applies S-box substitution to each byte of a 4-byte word

def sub_word(word: List[int]) -> List[int]:

    return [Sbox[b] for b in word]



# Performs a left cyclic shift of the word (e.g., [a,b,c,d] → [b,c,d,a])

def rot_word(word: List[int]) -> List[int]:

    return word[1:] + word[:1]



# Performs a left cyclic shift of the word

def key_expansion(key: bytes) -> List[List[int]]:

    """Expand 16-byte key into 44 4-byte words (AES-128)."""

    if len(key) != 16:
        raise ValueError("AES-128 key must be 16 bytes")

    key_symbols = list(key)

    w = [key_symbols[i:i+4] for i in range(0, 16, 4)]

    for i in range(4, 44):

        temp = w[i-1][:]

        if i % 4 == 0:

            temp = sub_word(rot_word(temp))

            temp[0] ^= Rcon[i//4]

        w.append([a ^ b for a, b in zip(w[i-4], temp)])

    return w  # 44 words



# AES processes data in a 4×4 state matrix

def add_round_key(state, round_key_words):

    for r in range(4):

        for c in range(4):

            state[r][c] ^= round_key_words[c][r]


# XORs the state matrix with the round key

def sub_bytes(state):

    for r in range(4):

        for c in range(4):

            state[r][c] = Sbox[state[r][c]]


# Replaces each byte in the state using the S-box
```

```
def shift_rows(state):

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

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

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


    # Row 0: no shift

    # Row 1: shift by 1

    # Row 2: shift by 2

    # Row 3: shift by 3



def xtime(a: int) -> int:

    return ((a << 1) ^ 0x1B) & 0xFF if (a & 0x80) else (a << 1) & 0xFF


def mix_columns(state):

    for c in range(4):

        a, b, d, e = state[0][c], state[1][c], state[2][c], state[3][c]

        t = a ^ b ^ d ^ e

        state[0][c] ^= t ^ xtime(a ^ b)

        state[1][c] ^= t ^ xtime(b ^ d)

        state[2][c] ^= t ^ xtime(d ^ e)

        state[3][c] ^= t ^ xtime(e ^ a)

# Mixes each column by performing a fixed polynomial multiplication, providing diffusion
```

```
def _bytes_to_state(block16: bytes):
    state = [[0]*4 for _ in range(4)]
    for r in range(4):
        for c in range(4):
            state[r][c] = block16[r + 4*c]
    return state
    # Converts a 16-byte block into a 4×4 state matrix (column-major order)


def _state_to_bytes(state) -> bytes:
    out = bytearray(16)
    idx = 0
    for c in range(4):
        for r in range(4):
            out[idx] = state[r][c]
            idx += 1
    return bytes(out)
    # Converts the 4×4 state back into a 16-byte sequence


def aes128_encrypt_block(block16: bytes, key: bytes) -> bytes:
    """Encrypt one 16-byte block."""
    if len(block16) != 16:
        raise ValueError("Block must be 16 bytes")
    w = key_expansion(key)
    state = _bytes_to_state(block16)
    add_round_key(state, w[0:4])


    for rnd in range(1, 10):
        sub_bytes(state)
        shift_rows(state)
        mix_columns(state)
        add_round_key(state, w[4*rnd:4*(rnd+1)])


    sub_bytes(state)
    shift_rows(state)
    add_round_key(state, w[40:44])
    return _state_to_bytes(state)


# Encrypts one 16-byte block using AES-128
# Steps:
# Key expansion to get all round keys
# Initial AddRoundKey
# 9 rounds of:
#     SubBytes → ShiftRows → MixColumns → AddRoundKey
# Final (10th) round:
#     SubBytes → ShiftRows → AddRoundKey (no MixColumns)
# Returns 16-byte encrypted block
# -------------------- CTR Mode --------------------

def _counter_block(nonce8: bytes, counter: int) -> bytes:

    if len(nonce8) != 8:

        raise ValueError("CTR nonce must be 8 bytes")

    return nonce8 + counter.to_bytes(8, "big")

    # Combines an 8-byte nonce with an 8-byte counter to form a 16-byte block



def aes128_ctr_crypt(data: bytes, key: bytes, nonce8: bytes, initial_counter: int = 0) ->
bytes:

    """Encrypt/Decrypt data using AES-128 in CTR mode."""

    out = bytearray()

    counter = initial_counter

    for i in range(0, len(data), 16):

        block = data[i:i+16]

        ks = aes128_encrypt_block(_counter_block(nonce8, counter), key)

        counter += 1

        out.extend(bytes(b ^ k for b, k in zip(block, ks[:len(block)])))

    return bytes(out)

# Splits the input into 16-byte chunks.

# For each chunk:

#     Generate keystream = AES(key, nonce || counter).

#     XOR keystream with plaintext (or ciphertext) to encrypt/decrypt.

#     Increment counter.
# Same function is used for encryption and decryption.


def encrypt_ctr(plaintext: bytes, key: bytes, nonce8: bytes) -> bytes:
    return aes128_ctr_crypt(plaintext, key, nonce8)


def decrypt_ctr(ciphertext: bytes, key: bytes, nonce8: bytes) -> bytes:
    return aes128_ctr_crypt(ciphertext, key, nonce8)
# Since CTR is symmetric, both encryption and decryption are identical
# -------------------- Demo --------------------
if __name__ == "__main__":
    key = b"MySecretAESKey!!"      # 16 bytes key
    nonce = b"\x00\x01\x02\x03\x04\x05\x06\x07"  # 8-byte nonce
    msg = b"CTR mode lets AES handle multiple blocks easily. No padding needed!"
    ct = encrypt_ctr(msg, key, nonce)
    pt = decrypt_ctr(ct, key, nonce)
    print("Plaintext :", msg)
    print("Ciphertext:", ct.hex())
    print("Decrypted :", pt)
    assert pt == msg
```

Result

```
Plaintext : b'CTR mode lets AES handle multiple blocks easily. No padding needed!'
Ciphertext:
4db3b2623f0a5148ea1f7ea692e53bc6cbe27e781a335c6e973a2527b1e789609ac829f6b5
2c73a7c0c2f8b6482f58ea184111985da6e242c35704fc969455f4f5e1a5
Decrypted : b'CTR mode lets AES handle multiple blocks easily. No padding needed!'
```

7. Observation Table

a. The plaintext was successfully encrypted into ciphertext using the AES-128 algorithm and a symmetric key in CTR (Counter) mode.
b. The same symmetric key and nonce were required to decrypt the ciphertext back to the original plaintext.
c. Unlike traditional block modes, CTR mode converts AES into a stream cipher, allowing encryption of variable-length messages without padding.
d. The strength of encryption depended on the secrecy of both the key and nonce, not the algorithm itself.
e. AES operated on 128-bit blocks with a 128-bit key size, and the encryption/decryption process was verified by matching the input plaintext with the decrypted output.

8. Conclusion

a. The implementation of AES-128 (Advanced Encryption Standard) in C++ demonstrates the practical working of modern symmetric key cryptography.
b. It ensures confidentiality and integrity of data as long as the secret key and nonce are securely shared between sender and receiver.
c. Compared to older standards like DES, AES provides stronger security due to its larger key size and resistance to brute-force attacks.
d. The experiment helped in understanding how block ciphers, key expansion, and counter mode operations work together to provide secure encryption and decryption, while eliminating the need for padding and ensuring efficient processing of large messages.

9. Resources

Sr. No.	Source	Link
1.	GeeksforGeeks	https://www.geeksforgeeks.org/computer-networks/advanced-encryption-standard-aes/