Experiment-1

AIM

Write a program that contains a string (char pointer) with a value “Hello world“. The program should XOR each character in this string with 0 and display the result.

PROGRAM

```
#include <stdio.h>

int main() {
    // Initialize the string char str[]
    = "Hello world";

    // XOR each character in the string with 0 for (int i = 0;
    str[i] != '\0'; i++) {
        str[i] ^= 0; // XOR with 0 (which has no effect on the character)
    }

    // Display the result
    printf("Resulting string: %s\n", str);
    return 0;
}
```

OUTPUT

```
C:\Users\styar\CNS_LAB>gcc exp1.c

C:\Users\styar\CNS_LAB>a.exe exp1.c
Resulting string: Hello world
```

DETAILED ANALYSIS

The provided code initialises a string "Hello world", performs the XOR operation on each character (which has no effect since XORing with 0 leaves characters unchanged), and prints the result.

The code explains that strings in C are arrays of characters terminated by a null character ('\0'). The XOR operation (^) is a bitwise operator that compares corresponding bits of two operands. When a character is XORed with 0, it remains unchanged since any value XORed with 0 yields the original value. This operation is used in the provided program to demonstrate the concept, even though it has no effect on the string.

Experiment-2

AIM

Write a C program that contains a string (char pointer) with a value „Hello world‟. The program should AND or and XOR each character in this string with 127 and display the result.

PROGRAM

```
#include <stdio.h>

int main() {
    // Initialize the string char str[]
    = "Hello world";

    // Create copies of the original string for AND and XOR operations char
    and_result[sizeof(str)]; char xor_result[sizeof(str)];

    // Perform AND and XOR operations with 127
    for (int i = 0; str[i] != '\0'; i++) {
        and_result[i] = str[i] & 127; // AND with 127
        xor_result[i] = str[i] ^ 127; // XOR with 127
    }

    // Null-terminate the result strings and_result[sizeof(str) - 1] =
    '\0'; xor_result[sizeof(str) - 1] = '\0';

    // Display the results printf("Original string: %s\n", str);
    printf("Result after AND with 127: %s\n", and_result);
    printf("Result after XOR with 127: %s\n", xor_result); return 0;
}
```

OUTPUT

```
C:\Users\styar\CNS_LAB>gcc exp2.c

C:\Users\styar\CNS_LAB>a.exe exp2.c
Original string: Hello world
Result after AND with 127: Hello world
‼esult after XOR with 127: 7→!!!!►►
```

DETAILED ANALYSIS

The blog post provides a C program that XORs each character of the string "HelloWorld" with 127 and also ANDs each character with 127. The resulting characters from both operations are printed.

Key Concepts:

<u>String</u>

- A one-dimensional array of characters terminated by a null character ('\0').

<u>AND Operation</u>

- Produces 1 only if both bits are 1.
- Performs bitwise AND with 127 on each character.
- `127` in binary is `01111111`, so AND operation retains the lower 7 bits, effectively no change for ASCII characters.

<u>XOR Operation</u>

- Produces 1 if the bits are different.
- Performs bitwise XOR with 127 on each character.
- XOR with 127 flips the highest bit and inverts the lower 7 bits.

Experiment-3

AIM

Write a program(s) to perform encryption and decryption using the following algorithms

1. Caesar Cipher
2. Substitution Cipher
3. Hill Cipher
4. Play fair Cipher

CAESAR CIPHER <u>Program</u>

```
def caeser(st,s):
   result="" for i in
   range(len(st)): char=st[i]
   if (char.isupper()):
         result += chr((ord(char) + s-65) % 26 + 65) else:
         result += chr((ord(char) + s - 97) % 26 + 97)
   return result

st="HELLO" s=3 print ("Text       : " +
st) print ("Shift : " + str(s)) print
("Cipher: " + caeser(st,s))
```

Output

```
:\Users\styar\CNS_LAB>python exp3_caeser.py
Text  : HELLO
Shift : 3
Cipher: KHOOR
```

Detailed Analysis

- Input:
  - A string st and an integer s (shift value).
- Initialization:
  - Create an empty string result to store the encrypted message.
- Processing:
  - For each character char in st:
    - If the character is uppercase:
- Calculate the new character using the formula:
  - chr((ord(char) - 65 + s) % 26 + 65) ● Append the new character to the result.
    - If the character is lowercase:
- Calculate the new character using the formula:
  - chr((ord(char) - 97 + s) % 26 + 97) ● Append a new character to the result.
- Output:
  - Return the result string containing the encrypted message.

Example: For the string "HELLO" and shift value 3-

- H becomes K ● E becomes H
- L becomes O ● L becomes O
- O becomes R

Result: "KHOOR"

Applications

- Simple encryption for educational purposes.
- Historical use by Julius Caesar.
- Basic data obfuscation.
- Puzzle games to introduce cryptographic principles.

SUBSTITUTION CIPHER

Program import
string import
random

```
def monoalphabetic_cipher(text, cipher_key=None, decrypt=False): alphabet =
    list(string.ascii_lowercase)

    if cipher_key is None: shuffled_alphabet = alphabet[:]
        random.shuffle(shuffled_alphabet) cipher_key =
        dict(zip(alphabet, shuffled_alphabet))
    else:
        if decrypt: cipher_key = {v: k for k, v in cipher_key.items()}

    translated_text = ''.join(cipher_key.get(char, char) for char in text.lower()) return translated_text,
    cipher_key


plaintext = "hello world"

# Encrypt the plaintext
encrypted_text, cipher_key = monoalphabetic_cipher(plaintext)
# Decrypt the encrypted text
decrypted_text, _ = monoalphabetic_cipher(encrypted_text, cipher_key, decrypt=True)

print(f"Cipher Key: {cipher_key}") print(f"Plaintext:
{plaintext}") print(f"Encrypted: {encrypted_text}")
print(f"Decrypted: {decrypted_text}")
```

Output

```
C:\Users\styar\CNS_LAB>python exp3_subs.py
Cipher Key: {'a': 'd', 'b': 't', 'c': 'u', 'd': 'g', 'e': 'n', 'f': 'f', 'g': 'l', 'h': 'm', 'i': 'q', 'j': 'p', 'k':
 'e', 'l': 'y', 'm': 'x', 'n': 'v', 'o': 'j', 'p': 'r', 'q': 'a', 'r': 'i', 's': 'z', 't': 'c', 'u': 's', 'v': 'o', '
w': 'h', 'x': 'k', 'y': 'w', 'z': 'b'}
Plaintext: hello world
Encrypted: mnyyj hjiyg
Decrypted: hello world
```

Detailed Analysis

1. Input:
    - A string text to be encrypted or decrypted.
    - An optional cipher key dictionary for character substitution. ○ A boolean decrypt flag to indicate decryption mode.
2. Initialization:
    - Define alphabet as the list of lowercase letters.

3. Cipher Key Generation:
    - If the cipher key is not provided:
        - Shuffle the alphabet to create a substitution cipher key.
        - Create a dictionary cipher key mapping original letters to shuffled letters.
    - If the cipher key is provided and decrypt is True:
        - Invert the cipher_key to map shuffled letters back to the original letters.
4. Text Translation:
    - For each character in the text:
        - If the character is in the cipher key, replace it with the corresponding character from the cipher key.
        - If the character is not in the cipher key, leave it unchanged. ○ Append the translated characters to the translated text.
5. Output:
    - Return the translated text and cipher key.

Example

Plaintext: "hello world" Encryption:

1. Generate a shuffled alphabet (e.g., {'a': 'x', 'b': 'm', 'c': 'l', ..., 'z': 'q'}).
2. Encrypt the plaintext using the cipher key:
    - "hello world" might become "xubbe mehbt" with the given example key.

Decryption:

1. Invert the cipher key.
2. Decrypt the encrypted text using the inverted key: ○ "xubbe mehbt" becomes "hello world".

Applications

1. Historical Encryption: Used in classical ciphers to encode messages.
2. Educational Tool: Demonstrates basic principles of substitution ciphers and cryptographic methods.
3. Puzzles and Games: Commonly used in escape rooms and puzzles for encoding clues.
4. Simple Data Obfuscation: Provides a basic layer of obfuscation for non-sensitive information.

HILL CIPHER Program

```
keyMatrix = [[0]*3 for i in range(3)] messageVector =
[[0] for i in range(3)] cipher = [[0] for i in range(3)]

def keyValue(key): k =
    0 for i in range(3):
```

```
        for j in range(3): keyMatrix[i][ j] =
            ord(key[k]) % 65 k+= 1

def encrypt(messageVector):
    for i in range(3):
        for j in range(1): cipher[i][ j] = 0 for x in range(3): cipher[i][ j]+=
            (keyMatrix[i][x] * messageVector[x][ j]) cipher[i][ j] = cipher[i][ j]
            % 26

def hillCipher(message, key): keyValue(key) for i in
    range(3): messageVector[i][0] = ord(message[i]) % 65
    encrypt(messageVector) cipherText = [] for i in range(3):
    cipherText.append(chr(cipher[i][0] + 65))
    print("Cipher Text= ", "".join(cipherText))

message = "SHR" key =
"HGFDSAEWQ"

hillCipher(message, key)
```

Output

```
C:\Users\styar\CNS_LAB>python exp3_hills.py
Cipher Text=  TYE
```

Detailed Analysis

1. Input:
    - A 3-character string `message`. ● A 9-character string `key`.

2. Initialization:
    - Define `keyMatrix` as a 3x3 matrix of zeros.
    - Define `messageVector` as a 3x1 matrix of zeros. ● Define `cipher` as a 3x1 matrix of zeros.

3. Key Matrix Generation:
    - Function `keyValue(key)`:
        - Convert each character of `key` to its numerical value (A=0, B=1, ..., Z=25).
        - Fill the `keyMatrix` with these values in row-major order.

4. Message Vector Generation:
   - Convert each character of `message` to its numerical value. ● Fill the `messageVector` with these values.

5. Encryption:
   - Function `encrypt(messageVector)`:
     - Multiply the `keyMatrix` by `messageVector`.
     - Take the result modulo 26 to get the `cipher` matrix.
6. Output Cipher Text:
- Convert numerical values in `cipher` back to characters. ● Concatenate these characters to form the cipher text.

Example
Input:
- Message: "SHR"
- Key: "HGFDSAEWQ"

Process:
1. Key Matrix:

HGFDSAEWQ -> Numerical values: `[7, 6, 5, 3, 18, 4, 22, 16]` keyMatrix:

[7 6 5
3 18 4
22 16 17]

2. Message Vector:

SHR -> Numerical values: `[18, 7, 17]` messageVector:

[18
7
17]

3. Encryption:

Multiply `keyMatrix` by `messageVector` and take modulo 26:

[7 6 5] [18] [3]
3 18 4] * [7] % 26 = [22]
[22 16 17] [17] [23] Cipher:
["D", "W", "X"]

Output:
- Cipher Text: "DWX"

Applications:

1. Military Communication: Used historically for secure military messages.
2. Cryptography Education: Demonstrates matrix multiplication and modular arithmetic.
3. Data Security: Provides a foundation for understanding more complex encryption methods.
4. Puzzle Solving: Used in cryptographic challenges and puzzles.

PLAYFAIR CIPHER Program def

```
generate_key_matrix(key): key =
key.replace(" ", "").upper() key =
key.replace("J", "I") matrix = []
used_chars = set()

    for char in key:
        if char not in used_chars: matrix.append(char)
            used_chars.add(char)

    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if char not in used_chars: matrix.append(char)
    used_chars.add(char) return [matrix[i:i+5] for i in
    range(0, 25, 5)]

def find_position(char, matrix):
    for i, row in enumerate(matrix): if char
        in row:
            return i, row.index(char)
    return None

def process_digraphs(text):
    text = text.replace(" ", "").upper().replace("J", "I")
    digraphs = [] i = 0 while i < len(text): a =
    text[i] b = text[i+1] if i+1 < len(text) else 'X' if
    a == b: digraphs.append(a + 'X') i += 1 else:
            digraphs.append(a + b) i +=
    2 if len(digraphs[-1]) == 1:
    digraphs[-1] += 'X'
    return digraphs

def playfair_encrypt(plaintext, key): matrix =
    generate_key_matrix(key) digraphs =
    process_digraphs(plaintext) ciphertext = ""
```

```
    for digraph in digraphs: a_row, a_col =
        find_position(digraph[0], matrix) b_row, b_col =
        find_position(digraph[1], matrix)

        if a_row == b_row: ciphertext += matrix[a_row][(a_col
            + 1) % 5] ciphertext += matrix[b_row][(b_col + 1) %
            5]
        elif a_col == b_col: ciphertext += matrix[(a_row + 1) %
        5][a_col] ciphertext += matrix[(b_row + 1) % 5][b_col]
        else:
            ciphertext += matrix[a_row][b_col] ciphertext +=

    matrix[b_row][a_col] return ciphertext

def playfair_decrypt(ciphertext, key): matrix =
    generate_key_matrix(key) digraphs =
    process_digraphs(ciphertext) plaintext = ""

    for digraph in digraphs: a_row, a_col =
        find_position(digraph[0], matrix) b_row, b_col =
        find_position(digraph[1], matrix)

        if a_row == b_row: plaintext += matrix[a_row][(a_col
            - 1) % 5] plaintext += matrix[b_row][(b_col - 1) %
            5]
        elif a_col == b_col: plaintext += matrix[(a_row - 1) %
        5][a_col] plaintext += matrix[(b_row - 1) % 5][b_col]
        else:
            plaintext += matrix[a_row][b_col] plaintext +=

    matrix[b_row][a_col] return plaintext

plaintext = "HELLO WORLD" key =
"KEYWORD"

encrypted_text = playfair_encrypt(plaintext, key) decrypted_text =
playfair_decrypt(encrypted_text, key)

print(f"Plaintext: {plaintext}") print(f"Encrypted:
{encrypted_text}") print(f"Decrypted: {decrypted_text}")
```

Output

```
C:\Users\styar\CNS_LAB>python exp3_playfair_2.py
Plaintext: HELLO WORLD
Encrypted: GYIZSCOKCFBU
Decrypted: HELXLOWORLDX
```

Detailed Analysis

1. Input: A plaintext string and a keyword.

2. Generate Key Matrix:
   - Remove spaces and convert the key to uppercase.
   - Replace 'J' with 'I'.
   - Create a 5x5 matrix using unique characters from the key followed by the remaining letters of the alphabet (excluding 'J').

3. Process Digraphs:
   - Remove spaces and convert the plaintext to uppercase.
   - Replace 'J' with 'I'.
   - Form digraphs (pairs of letters). If a pair consists of the same letter, insert an 'X' between them. If there's an odd number of letters, append 'X' to the last letter.
4. Encrypt:
   - For each digraph:
     - If both letters are in the same row, replace them with the letters to their right (wrap around if needed).
     - If both letters are in the same column, replace them with the letters below (wrap around if needed).
     - If they form a rectangle, swap their columns.

5. Decrypt:
   - For each digraph:
     - If both letters are in the same row, replace them with the letters to their left (wrap around if needed).
     - If both letters are in the same column, replace them with the letters above (wrap around if needed).
     - If they form a rectangle, swap their columns.

6. Output: Return the encrypted or decrypted text.

Example
Input:

- Plaintext: "HELLO WORLD"
- Key: "KEYWORD"

Key Matrix:

K E Y W O
R D A B C
F G H I L
M N P Q S T
U V X Z

Digraphs:

- Original: "HELLOWORLD"
- Processed: "HE LX LO WO RL DX"

Encryption Steps:

- "HE" -> "KA"
- "LX" -> "AT"
- "LO" -> "WR"
- "WO" -> "OW"
- "RL" -> "DR"
- "DX" -> "XA"

Encrypted Output:

- Encrypted Text: "KATWRWDRXA"

Decryption:

- Following the same process, you can retrieve the original plaintext.

Applications of the Playfair Cipher

1. Historical Use: Used in World War I and II for secure military communications.
2. Cryptography Education: Teaches foundational concepts of encryption and cryptography.
3. Puzzle Creation: Used in cryptographic puzzles and escape room challenges.
4. Data Security: Provides a basic level of security for sensitive communications.
5. Modern Variations: Forms the basis for more complex encryption techniques.

Experiment-4

AIM
Write a program to implement the DES algorithm logic.

PROGRAM

```
from Crypto.Cipher import DES from
Crypto.Util.Padding import pad, unpad

# Function to encrypt plaintext using DES def
encrypt(plaintext, key):
   # Create a DES cipher object cipher =
   DES.new(key, DES.MODE_CBC)

   # Pad the plaintext to be a multiple of block size padded_text =
   pad(plaintext.encode(), DES.block_size)

   # Encrypt the padded plaintext ciphertext = cipher.encrypt(padded_text) return cipher.iv,

   ciphertext     # Return initialization vector and ciphertext

# Function to decrypt ciphertext using DES def
decrypt(ciphertext, key, iv): # Create a DES cipher
object cipher = DES.new(key, DES.MODE_CBC, iv)

   # Decrypt the ciphertext
   decrypted_padded_text = cipher.decrypt(ciphertext)

   # Unpad the decrypted text
   decrypted_text = unpad(decrypted_padded_text, DES.block_size) return

   decrypted_text.decode()       # Return the decrypted plaintext


key = b'abcdefgh'             # Key must be 8 bytes long

plaintext = "Hello, World!"
   # Encrypt the plaintext iv, ciphertext =
encrypt(plaintext, key)

   # Display the results
```

print(f"Plaintext: {plaintext}") print(f"Ciphertext (hex):
{ciphertext.hex()}") print(f"Initialization Vector (IV):
{iv.hex()}")

# Decrypt the ciphertext decrypted_text =
decrypt(ciphertext, key, iv)

Display the decrypted text print(f"Decrypted text:
{decrypted_text}")

OUTPUT

```
C:\Users\styar\CNS_LAB>python exp8_des.py
Plaintext: Hello, World!
Ciphertext (hex): 819459e5b910ed07523aa825cbc75ecf
Initialization Vector (IV): 198a7ec18694af08
Decrypted text: Hello, World!
```

DETAILED ANALYSIS

The DES algorithm operates on 64-bit blocks of data and uses a 56-bit key. Here are the detailed steps:

1. Initial Permutation (IP):
   - The 64-bit plaintext block undergoes an initial permutation that rearranges the bits.
2. Key Schedule Generation:
   - The 56-bit key is divided into two 28-bit halves.
   - Each half is then shifted (rotated) left by one or two positions (depending on the round).
   - 16 subkeys, each 48 bits long, are generated for the 16 rounds of the algorithm.
3. 16 Rounds of Feistel Structure:
   - The permuted block is divided into two halves: left (L) and right (R). ○ For each of the 16 rounds, the following operations are performed:
     - Expansion (E): The 32-bit right half (R) is expanded to 48 bits using the expansion permutation.
     - Key Mixing: The expanded R is XORed with the round's subkey.
     - Substitution (S-boxes): The 48-bit result is divided into eight 6-bit blocks. Each block is substituted using a predefined 6x4 S-box, resulting in a 32-bit output.
     - Permutation (P): The 32-bit output of the S-boxes is permuted.
     - Function Output: The permuted result is XORed with the left half (L).
     - The left half (L) is then swapped with the right half (R) for the next round.
4. Final Permutation (FP):

- After 16 rounds, the final left and right halves are concatenated and subjected to a final permutation, which is the inverse of the initial permutation.

Example of DES Encryption
Let’s consider a simple example to illustrate the DES encryption process:

- Plaintext: 0123456789ABCDEF (64-bit block in hexadecimal)
- Key: 133457799BBCDFF1 (64-bit block in hexadecimal, 56 bits used for key schedule)

Steps:
1. Initial Permutation (IP):
   - Apply the initial permutation to the plaintext to get a rearranged 64-bit block.
2. Key Schedule Generation:
   - Generate 16 subkeys from the 56-bit key.
3. 16 Rounds of Feistel Structure:
   - Perform 16 rounds of the Feistel structure using the subkeys and intermediate data.
4. Final Permutation (FP):
   - Apply the final permutation to the concatenated result of the 16th round.

The output will be a 64-bit ciphertext block

Applications of DES
1. Historical Data Encryption: DES was widely used for encrypting sensitive data in financial transactions, government communications, and secure data storage.
2. Legacy Systems: Some older systems still use DES due to legacy compatibility requirements.
3. Educational Purposes: DES is often taught in cryptography courses to illustrate the principles of block ciphers and symmetric-key encryption.
4. Triple DES (3DES): To address the security limitations of DES, Triple DES (3DES) was developed, which applies the DES algorithm three times with different keys, enhancing its security.

Experiment-5

AIM
Write a program to implement the Blowfish algorithm logic.

PROGRAM
```
from Crypto.Cipher import Blowfish from
Crypto.Util.Padding import pad, unpad

# Function to encrypt a message
def encrypt_blowfish(key, plaintext):
    cipher = Blowfish.new(key, Blowfish.MODE_ECB) padded_text =
    pad(plaintext.encode(), Blowfish.block_size) encrypted_text =
    cipher.encrypt(padded_text) return encrypted_text

# Function to decrypt a message def
decrypt_blowfish(key, encrypted_text):
    cipher = Blowfish.new(key, Blowfish.MODE_ECB) decrypted_padded_text =
    cipher.decrypt(encrypted_text) decrypted_text = unpad(decrypted_padded_text,
    Blowfish.block_size) return decrypted_text.decode()

# Example usage
key = b'Sixteen byte key'           # Key must be between 4 and 56 bytes message
= "Hello world" print("Original message:", message)

encrypted_message = encrypt_blowfish(key, message) print("Encrypted message:",
encrypted_message)

decrypted_message = decrypt_blowfish(key, encrypted_message) print("Decrypted message:",
decrypted_message)
```

OUTPUT
```
C:\Users\styar\CNS_LAB>python exp7_blowfish.py
Original message: Hello world
Encrypted message: b'\xb1\xa1\xf1\x0b\x01 \xbd\xd3g\xf7\xf1pJS\xc2a'
Decrypted message: Hello world
```

DETAILED ANALYSIS
The Blowfish algorithm is known for its simplicity, speed, and security, and is widely used for encrypting data. Blowfish operates on 64-bit blocks and uses a variable-length key ranging from 32 bits to 448 bits.

Steps of the Blowfish Algorithm

1. Key Expansion:
    - The key expansion phase converts a key of up to 448 bits into several subkey arrays totaling 4168 bytes.
    - It generates 18 32-bit subkeys (P-array) and four 32-bit S-boxes (each with 256 entries).
2. Data Encryption:
    - Blowfish uses a 16-round Feistel network.
    - Each round consists of a key-dependent permutation and a keyand data-dependent substitution.
    - The algorithm splits the 64-bit block into two 32-bit halves and then iterates through 16 rounds of encryption.
    - The two halves are combined and subjected to a final permutation to produce the ciphertext.

Example:
Initialization

- Plaintext: "Hello world"
- Key: "simplekey" (in hexadecimal for clarity, but in practice, keys can be up to 448 bits long)

Key Expansion:

- The key "simplekey" is used to generate the P-array and S-boxes.

Padding:

- The plaintext "Hello world" is padded to ensure its length is a multiple of the block size (8 bytes for Blowfish). This padding ensures the algorithm can process the data in fixed-size blocks.

Encryption:

- The padded plaintext is encrypted using the Blowfish cipher in ECB (Electronic Codebook) mode, producing the ciphertext.

Decryption:

- The ciphertext is decrypted back to the padded plaintext.
- The padding is removed to retrieve the original plaintext "Hello world".

Applications of Blowfish

1. Secure File Transfer: Blowfish can encrypt files before they are transferred over insecure channels, ensuring data confidentiality.
2. Password Protection: Blowfish is used in password hashing algorithms like bcrypt, which adds a salt and makes it computationally expensive to perform brute-force attacks.
3. VPNs and Network Security: Blowfish is used to encrypt data transmitted over Virtual Private Networks (VPNs) and secure network communications.

4. Disk Encryption: Some disk encryption software uses Blowfish to protect data stored on hard drives.
5. Database Security: Blowfish can be used to encrypt sensitive information in databases, ensuring that data remains secure even if the database is compromised.

Experiment-6

AIM

Write a program to implement the Rijndael algorithm logic.

PROGRAM

```
from Crypto.Cipher import AES from Crypto.Util.Padding
import pad, unpad from Crypto.Random import
get_random_bytes

# Function to encrypt a message def
encrypt_aes(key, plaintext):
cipher = AES.new(key, AES.MODE_CBC)            # Using CBC mode
    iv = cipher.iv  # Initialization vector padded_text =
    pad(plaintext.encode(), AES.block_size) encrypted_text =
    cipher.encrypt(padded_text) return iv + encrypted_text

# Function to decrypt a message def decrypt_aes(key, encrypted_text): iv =
encrypted_text[:AES.block_size]# Extract the IV from the beginning cipher = AES.new(key,
AES.MODE_CBC, iv)
    decrypted_padded_text = cipher.decrypt(encrypted_text[AES.block_size:]) decrypted_text =
    unpad(decrypted_padded_text, AES.block_size) return decrypted_text.decode()

# Example usage
key = get_random_bytes(16)      # AES-128 key
message = "Hello world" print("Original message:",
message)

encrypted_message = encrypt_aes(key, message) print("Encrypted message:",
encrypted_message) decrypted_message = decrypt_aes(key, encrypted_message)
print("Decrypted message:", decrypted_message)
```

OUTPUT

```
C:\Users\styar\CNS_LAB>python exp9_aes.py
Original message: Hello world
Encrypted message: b'\xa6~7G\xd7a\xc2T|Y\xb3\xbd\t:/z\x1aU\xc9#\xb6,\x10\xbf\x84\xab\xa1\xed\xb6?\x1f\xc8'
Decrypted message: Hello world
```

DETAILED ANALYSIS

The Advanced Encryption Standard (AES) is a symmetric key encryption algorithm that has become the standard for securing data. It is based on the Rijndael cipher. AES operates on fixed-size blocks of data (128 bits) and supports key sizes of 128, 192, and 256 bits.

AES Algorithm Steps 1.
Key Expansion:
- The original encryption key is expanded into an array of key schedule words (subkeys) for each round.

2. Initial Round:
   - AddRoundKey: The initial state is XORed with the first round key.
3. Main Rounds (for 128-bit key, there are 10 rounds):
   - Each round consists of four transformations:
     1. SubBytes: Each byte in the state is substituted with another byte using an S-box (substitution box).
     2. ShiftRows: Each row in the state is shifted left by a certain number of bytes.
     3. MixColumns: Each column in the state is mixed to provide diffusion.
     4. AddRoundKey: The state is XORed with the round key derived from the key schedule.
4. Final Round:
   - This round consists of the first three transformations (SubBytes, ShiftRows, AddRoundKey), but without the MixColumns step.

Example of AES Encryption
Let's illustrate the AES encryption process with a simple example:
- Plaintext: "Two One Nine Eight" (in hexadecimal: 54676f20576f6e65204e696e652038) which is 128 bits.
- Key: "Thats my Kung Fu" (in hexadecimal: 5468617473206d79204b756e67204675) which is 128 bits.

AES Encryption Steps
1. Key Expansion:
   - Expand the key into a set of round keys.
2. Initial Round:
   - AddRoundKey: XOR the plaintext with the first round key.
3. Main Rounds:
   - For each round, perform the four transformations (SubBytes, ShiftRows, MixColumns, AddRoundKey).
4. Final Round:
   - Perform the last three transformations (SubBytes, ShiftRows, AddRoundKey).

Output:
Ciphertext (hex): 2c2d45f58f01511ff403b8d99daacccf

Initialization Vector (IV): 3c5e5b56cc7db709e9f6d64b018b2a2d Decrypted text: Two One Nine Eight

Applications:

1. Secure Communication: AES is used in securing communication protocols such as TLS (Transport Layer Security) and SSL (Secure Sockets Layer) for secure internet communications.
2. File Encryption: AES is commonly used to encrypt files and sensitive data on storage devices, ensuring data confidentiality.
3. Wireless Security: AES is employed in Wi-Fi Protected Access (WPA2) and WPA3 standards to secure wireless communications.
4. Database Encryption: AES is used to encrypt sensitive data stored in databases, protecting it from unauthorized access.
5. Disk Encryption: Full disk encryption solutions like BitLocker and FileVault use AES to protect data at rest on hard drives.
6. Digital Rights Management (DRM): AES is used in DRM systems to protect copyrighted digital content from unauthorized use.

Experiment-7

AIM

Write a program to implement the RC4 algorithm logic.

PROGRAM

```
def KSA(key):
    """Key Scheduling Algorithm for RC4"""
    key_length = len(key) S =
    list(range(256)) j = 0 for i
    in range(256):
        j = (j + S[i] + key[i % key_length]) % 256
        S[i], S[ j] = S[ j], S[i]# Swap return S

def PRGA(S):
    """Pseudo-Random Generation Algorithm for RC4"""
    i = j = 0 while True: i = (i + 1) %
    256 j = (j + S[i]) % 256 S[i], S[ j] =
    S[ j], S[i]        # Swap K = S[(S[i] +
    S[ j]) % 256] yield K

def rc4(key, plaintext):
    """RC4 encryption/decryption function""" key = [ord(k) for
    k in key]        # Convert key to ASCII S = KSA(key) keystream
    = PRGA(S)
    ciphertext = ''.join(chr(ord(p) ^ next(keystream)) for p in plaintext) return ciphertext

# Example usage key = "SecretKey"
plaintext = "Hello, World!" ciphertext
= rc4(key, plaintext)
print("Ciphertext:", ciphertext)

# To decrypt, run rc4 again with the same key and ciphertext
decrypted = rc4(key, ciphertext) print("Decrypted:",
decrypted)
```

OUTPUT

```
C:\Users\styar\CNS_LAB>python exp4_rc4.py
Ciphertext: \ÝP+°^>õaQ"
Decrypted: Hello, World!
```

DETAILED ANALYSIS

It is known for its simplicity and speed in software. RC4 generates a pseudo-random stream of bits (key stream) that is combined with plaintext to produce ciphertext through an XOR operation.

1. Key Scheduling Algorithm (KSA):
   - Initialize the state array SSS of size 256.
   - Populate SSS with the values from 0 to 255.
   - Use the secret key to permute the array SSS.
2. Pseudo-Random Generation Algorithm (PRGA):
   - Initialize two variables, iii and jjj, both set to 0.
   - Generate the key stream by swapping values in the state array and using them to produce the output stream.
3. Encryption/Decryption:
   - The plaintext is XORed with the generated key stream to produce ciphertext.
   - The ciphertext can be decrypted by XORing it again with the same key stream.

RC4 Algorithm Example

Let's illustrate the RC4 encryption process with a simple example:

- Key: Key
- Plaintext: Plaintext

1. Key Scheduling Algorithm (KSA):
   - Convert the key to ASCII values:
     - Key: Key → ASCII: [75, 101, 121] ○

     Initialize the state array SSS:
     - S=[0,1,2,…,255]S = [0, 1, 2, \ldots, 255]S=[0,1,2,…,255]
   - Key length L= 3 (for Key) ○

     For iii from 0 to 255:
     - j=(j+S[i]+Key[imod L])mod 256
     - Swap S[i]S[i]S[i] and S[j]S[j]S[j] 2.

   Pseudo-Random Generation Algorithm (PRGA):
   - Initialize iii and jjj to 0.
   - Generate the key stream:
     - For kkk from 0 to length of plaintext:
       - i=(i+1)mod 256
       - j=(j+S[i])mod 256
       - Swap S[i]S[i]S[i] and S[j]S[j]S[j] ■

         Output: K=S[(S[i]+S[j])mod 256]
3. Encryption:
   - XOR each byte of plaintext with the corresponding byte of the key stream to produce ciphertext.

Applications of RC4

1. SSL/TLS Protocols: RC4 was commonly used in older versions of SSL/TLS for securing web traffic.
2. WEP (Wired Equivalent Privacy): RC4 was used in the WEP protocol for securing wireless networks, although this has been largely replaced by more secure protocols like WPA2.
3. Secure File Transfer: RC4 was used in some file transfer protocols to encrypt files during transmission.
4. VPN Protocols: Some virtual private network (VPN) implementations utilized RC4 for encrypting data streams.
5. Streaming Protocols: RC4 was used in some streaming protocols due to its speed in encrypting data on the fly.
6. Encrypting Cookies: RC4 has been used to encrypt cookies in web applications to protect sensitive information.