Pabna University of Science and Technology

Faculty of Engineering and Technology

Department of Information and Communication Engineering

Lab report

Course Title: Cryptography and Computer Security Sessional.

Course code: ICE-4108

Submitted by:	Submitted to:
Name: Sumayta Shama Roll: 200639 Session: 2019-2020 4th year 1st semester. Department of Information and Communication Engineering, PUST.	Md. Anwar Hossain Professor, Department of Information and Communication Engineering, Pabna University of Science and Technology.

Submission date: 12-11-2024 Signature:

INDEX

Experiment No: 01

Experiment Name: Write a program to implement encryption and decryption using Caesar cipher.

Objective:

1. To implement the algorithm Caesar cipher for encryption and decryption of messages.
2. To learn about the concept of shifting character in any alphabet for the security of message
3. To Understand how cryptographic technique helps in securing messages.
4. To apply Caesar cipher on sample data and verify the results for encryption and decryption.

Theory: The Caesar cipher is one of the oldest and easiest forms of encryption. It is named after Julius Caesar, who used it for communicating securely. In this cipher, each letter in the plaintext is replaced by a letter some fixed number of positions down or up the alphabet. For example, with a shift of 3:

'A' becomes 'D'

'B' becomes 'E'

'C' becomes 'F'

In decryption, the process is reversed and the letters are shifted back to their original position. The shift value is used as a key for both the case of encryption and decryption.

Encryption Formula:

For each character in the plaintext, the new character is determined by:

$$C = (P + \text{shift}) \% 26$$

Here C= Cipher Character

P=Position of the plain text character

Shift= Fixed value by which to shift

Here modulus 26 represents the value should contain between the alphabet.

Decryption Formula:

$$P = (C - \text{shift}) \% 26$$

They represent the same meaning as used in encryption.

Example: If we take the message "SHAMA" and want to encrypt it with a shift of 3, then

S	H	A	M	A
V	K	D	P	D

So from the table it is seen that for the plain text message “SHAMA”, cipher text is “VKDPD”
Here we have added 3 to each of the letter of plain text, for decrypting it we simply have to use the decryption formula. For understanding it need to mention that all the letter are assigned unique numerical value for A to Z, 0 to 25.
For example, the numerical value of S is 18,
So (18+3) %26 is equal to 21, in which there is the letter V. This is how all the letter has been encrypted.

Algorithm:

For Encryption,

1. Choose a number as the "shift" value that decides how much each letter will move in the alphabet.
2. Start with the message and notice each letter in the message.
3.Move each letter forward by shifting each letter with the chosen shift value.
4.Save the new letter instead of the old one.
5.Continue doing this for every letter in the message.

For Decryption,

1. Take the encrypted message.
2. Find out the shift value that is used for encryption.
3. Check each letter in the encrypted message and move each letter back using the shift value and decrypt it. For example, if the shift value is 3, then "D" will become "A" like this.
4.Save the decrypted letter in position of the encrypted letter
5. Continue doing this for all the letters in the plain text encrypted message.

Code (Python):

```
def caesar_encrypt(plaintext, shift):
  encrypted_text = ""
  for char in plaintext:
    if char.isalpha():
      if char.islower():
        encrypted_text += chr((ord(char) + shift - ord('a')) % 26 + ord('a'))
      else:
        encrypted_text += chr((ord(char) + shift - ord('A')) % 26 + ord('A'))
    else:
      encrypted_text += char
  return encrypted_text
def caesar_decrypt(ciphertext, shift):
  decrypted_text = ""
  for char in ciphertext:
    if char.isalpha():
      if char.islower():
        decrypted_text += chr((ord(char) - shift - ord('a')) % 26 + ord('a'))
```

```
            else:
                decrypted_text += chr((ord(char) - shift - ord('A')) % 26 + ord('A'))
        else:
            decrypted_text += char
    return decrypted_text
plaintext = "SHAMA"
shift = 3
ciphertext = caesar_encrypt(plaintext, shift)
print(f"Caesar Cipher Encryption: {ciphertext}")
plaintext = caesar_decrypt(ciphertext, 3)
print(f"Caesar Cipher Decryption: {plaintext}")
```

Input: Caesar Cipher Encryption: VKDPD
Output: Caesar Cipher Decryption: SHAMA

Experiment No: 02

Experiment Name: Write a program to implement encryption and decryption using Mono-Alphabetic cipher.

Objective:

1. To know how Mono-Alphabetic substitution works by replacing each letter of the plaintext with another fixed letter.
2. To show the encryption and decryption processes with a fixed and predefined substitution key.

Theory: A Mono-Alphabetic cipher is a type of substitution cipher where each letter in the plaintext is mapped to a unique letter in the ciphertext alphabet. In Mono-Alphabetic ciphers, the substitution can be arbitrary and the mapping is constant throughout the whole encryption and decryption process. Mono Alphabetic cipher is far more secure than Caesar cipher because here for the cipher sequence permutation is used. The cipher line can by any permutation of the 26 letters so there are 26! Or greater than 4*10^26 possible keys.

But it relies on a fixed replacement structure that means substitution is fixed for each letter of the alphabet. Such as if A is once encoded as F, then every time it will be replaced as F. So, it can be guessed some time by the frequency of the letters. Even some time two or three letter is found out others become easy to know. Thinking about this side this technique can be called as a simple one.

So, the main encryption technique is taking each letter of the plaintext and replace it with the corresponding letter from the substitution key. In case of decryption, we need to take each letter of the ciphertext and replace it with the corresponding letter from the reverse substitution key.

Example, suppose we have the following substitution key:

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

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

So, if we want to encrypt ‘SHAMA’ then it will be encrypted like this,

S	H	A	M	A
L	I	Q	D	Q

The encrypted word is “LIQDQ”

Algorithm:

For encryption,

1. Take the input plaintext.
2. Define the substitution key that will be used which will be a fixed substitution cipher alphabet.

3. For each letter in the plaintext find the corresponding letter in the cipher alphabet and then replace it with the cipher letter.
4. Continue doing it for every letter in the plain text and then the cipher text will be found out.

For Decryption,

1. For each letter in the ciphertext find the corresponding letter in the plaintext alphabet and then replace it the ciphertext letter with the original letter.
2. Continue doing it for each letter of the cipher text and now the plain text will be decrypted.

Code (Python):

```
class MonoalphabeticCipher:
  def __init__(self):
    self.normal_char = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
                'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',
                's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

    self.coded_char = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O',
                'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K',
                'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M']

  def string_encryption(self, s):
    encrypted_string = ""
    for char in s:
      for i in range(26):
        if char == self.normal_char[i]:
          encrypted_string += self.coded_char[i]
          break
        elif char < 'a' or char > 'z':
          encrypted_string += char
          break
    return encrypted_string

  def string_decryption(self, s):
    decrypted_string = ""
    for char in s:
      for i in range(26):
        if char == self.coded_char[i]:
          decrypted_string += self.normal_char[i]
          break
        elif char < 'A' or char > 'Z':
          decrypted_string += char
          break
    return decrypted_string


def main():
  cipher = MonoalphabeticCipher()
  plain_text = "SHAMA"

  print("Plain text:", plain_text)

  # Changing the whole string to lowercase
  encrypted_message = cipher.string_encryption(plain_text.lower())
  print("Encrypted message:", encrypted_message)

  decrypted_message = cipher.string_decryption(encrypted_message)
  print("Decrypted message:", decrypted_message)



main()
```

Input: Plain text: SHAMA
Output: Encrypted message: LIQDQ
Decrypted message: SHAMA

Experiment No: 03
Experiment Name: Write a program to implement encryption and decryption using Brute force attack cipher.
Objective:
1.To understand the brute force technique for cryptographic analysis where all possible keys are used for trying to break the cipher.
2.To know how simple ciphers like the Caesar cipher is vulnerable to brute force attacks.
3.To show how a brute force attack can reveal the plaintext by trying every possible shift value in the Caesar cipher which obstruct it from being secure.

Theory: A brute force attack in cryptography is known as a method of breaking a cipher by systematically trying all possible keys until the correct one is found. This attack is found especially effective against weak encryption process like the Caesar cipher, where the number of possible keys is small.
In Caesar cipher which is a simple substitution technique each letter in the plaintext is shifted by a fixed number of positions. For example, with a shift of 3, 'A' becomes 'D', 'B' becomes 'E', and so on. Since there are only 26 letters in the English alphabet, the number of possible shift values is limited to 26 which is from 0 to 25. This makes the Caesar cipher highly vulnerable to brute force attacks.
In a brute force attack on the Caesar cipher, the attacker tries to decrypt the ciphertext by trying all possible shifts until the correct plaintext is found. Each shift is thought as a possible key, and the attacker looks for meaningful text in the output.
Example:
Suppose the ciphertext is "KHOOR" which is encrypted with a shift of 3 from the plaintext "HELLO". A brute force attack would try all possible shifts from 0 to 25:
Shift 0: KHOOR
Shift 1: JGNNQ
Shift 2: IFMMP
Shift 3: HELLO (Correct shift)
The correct shift (3) is identified when "HELLO" is revealed as the plaintext.

Algorithm:
1. Input the ciphertext, possible for the attacker when he has access to the encrypted message.

2.Iterate through all possible shifts which is from 0 to 25 for each shift.
3.Apply the shift to decrypt the ciphertext and check the resulting text.
4.Check for meaningful plaintext. if the attacker tries to find the correct shift by looking for meaningful words in the decrypted text.
5. Stop when the correct shift is found.

Code (Python):

```
# Function to decrypt the message with a given shift
def decrypt_caesar_cipher(ciphertext, shift):
    plaintext = ""
    for char in ciphertext:
        # Only decrypt alphabetic characters
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            decrypted_char = chr((ord(char) - shift_base - shift) % 26 + shift_base)
            plaintext += decrypted_char
        else:
            plaintext += char  # Non-alphabetic characters remain unchanged
    return plaintext

# Function to perform brute force attack on Caesar cipher
def brute_force_caesar(ciphertext):
    for shift in range(26):  # Trying all possible shifts (0-25)
        decrypted_message = decrypt_caesar_cipher(ciphertext, shift)
        print(f"Shift {shift}: {decrypted_message}")

# Input from user (ciphertext to be decrypted)
ciphertext = input("Enter the ciphertext to decrypt: ")

# Perform brute force attack
print("Attempting brute force attack on the ciphertext:")
brute_force_caesar(ciphertext)
```

Input: Enter the ciphertext to decrypt: KHOOR

Output: Attempting brute force attack on the ciphertext:

```
        Shift 0: KHOOR
        Shift 1: JGNNQ
        Shift 2: IFMMP
        Shift 3: HELLO
        Shift 4: GDKKN
        ...
        Shift 25: LIPPS
```

Experiment No: 04
Experiment Name: Write a program to implement encryption and decryption using Hill cipher.
Objective:
1.To implement encryption and decryption using the Hill cipher which is a polygraphic substitution cipher.
2.To understand matrix-based encryption in that process blocks of plaintext are multiplied by a key matrix to produce ciphertext.
3.To explore and analyze the use of linear algebra in cryptography, matrix multiplication and matrix inversion for encryption and decryption.
Theory:
Hill cipher is a polygraphic substitution cipher that encrypts blocks of letters using matrix multiplication which was invented by Lester S. Hill in 1929. It is an example of a cipher that operates on groups of letters together instead of doing it individually.
The cipher uses an invertible square matrix (2x2 or 3x3) as the encryption key. The size of the key matrix determines how many letters are grouped together for encryption. For example, a 2x2 matrix encrypts two letters at a time, while a 3x3 matrix can encrypt three letters at a time.
The plaintext is divided into blocks where each block can convert into a numerical vector. Each letter is replaced by its corresponding number (A=0, B=1, ..., Z=25).
Then plaintext vector is multiplied by the key matrix and the result is taken modulo 26 to produce the ciphertext vector. The ciphertext vector is then converted back to letters.
To decrypt the ciphertext, the inverse of the key matrix is used. The ciphertext vector is multiplied by the inverse matrix (mod 26) to recover the plaintext.

Example:
Example:
For encrypting the word "HI" using a 2x2 key matrix:
Key Matrix: [3 3]
[2 5]

Plaintext: "HI" -> [7, 8] (H = 7, I = 8)
Multiplying the matrix by the vector:
Ciphertext Vector = (Key Matrix) × (Plaintext Vector)
= [3 3] × [7] = [45]

[2 5]   [8]   [54]

Taking modulo 26:

Ciphertext Vector = [45 % 26, 54 % 26] = [19, 2]

Ciphertext: "TC" (T = 19, C = 2)

Algorithm:

For Encrypting,

1.Convert each letter of the plaintext to a numerical value: A = 0, B = 1, C = 2, ..., Z = 25.

2.Group the plaintext into blocks that match the size of the key matrix (e.g., 2-letter blocks for a 2x2 matrix).

3.Multiply each plaintext block (as a vector) by the key matrix.

4.Take the result modulo 26 to get the ciphertext vector.

5.Convert the resulting ciphertext vector back into letters.

For Decrypting,

1.Compute the inverse of the key matrix (mod 26).

2.Multiply the ciphertext vector by the inverse matrix.

3.Take the result modulo 26 to get the plaintext vector.

4.Convert the resulting vector back into letters.

Code (Python):

```
import numpy as np

# Function to convert letters to numbers (A=0, B=1, ..., Z=25)
def letters_to_numbers(plaintext):
    return [ord(char) - ord('A') for char in plaintext.upper()]

# Function to convert numbers back to letters
def numbers_to_letters(numbers):
    return ''.join([chr(num + ord('A')) for num in numbers])

# Function to encrypt using Hill cipher
def hill_cipher_encrypt(plaintext, key_matrix):
    plaintext_numbers = letters_to_numbers(plaintext)

    # Ensure plaintext length is a multiple of the matrix size
    block_size = key_matrix.shape[0]
    while len(plaintext_numbers) % block_size != 0:
        plaintext_numbers.append(ord('X') - ord('A'))  # Padding with 'X'

    plaintext_blocks = np.array(plaintext_numbers).reshape(-1, block_size)
    ciphertext_numbers = []

    # Encrypt each block
    for block in plaintext_blocks:
        cipher_block = np.dot(key_matrix, block) % 26
        ciphertext_numbers.extend(cipher_block)

    return numbers_to_letters(ciphertext_numbers)

# Function to decrypt using Hill cipher
def hill_cipher_decrypt(ciphertext, key_matrix):
    ciphertext_numbers = letters_to_numbers(ciphertext)

    block_size = key_matrix.shape[0]
    ciphertext_blocks = np.array(ciphertext_numbers).reshape(-1, block_size)

    # Compute the inverse matrix mod 26
    det = int(round(np.linalg.det(key_matrix)))
    det_inv = pow(det, -1, 26)  # Modular inverse of determinant mod 26
    adjugate_matrix = np.round(det * np.linalg.inv(key_matrix)).astype(int) % 26
    inverse_matrix = (det_inv * adjugate_matrix) % 26

    plaintext_numbers = []

    # Decrypt each block
    for block in ciphertext_blocks:
        plain_block = np.dot(inverse_matrix, block) % 26
        plaintext_numbers.extend(plain_block)

    return numbers_to_letters(plaintext_numbers)

# Example key matrix (2x2)
key_matrix = np.array([[3, 3], [2, 5]])

# Input from user
plaintext = input("Enter the plaintext (only letters): ")

# Encrypt the message
ciphertext = hill_cipher_encrypt(plaintext, key_matrix)
print(f"Encrypted message: {ciphertext}")

# Decrypt the message
decrypted_message = hill_cipher_decrypt(ciphertext, key_matrix)
print(f"Decrypted message: {decrypted_message}")
```

Input: Enter the plaintext (only letters): HI

Output: Encrypted message: TC

Decrypted message: HI

Experiment No: 05
Experiment Name: Write a program to implement encryption using Playfair cipher.
Objective:
1. To understand the working process of the Playfair cipher for secure message transmission.
2.To learn how a 5x5 matrix is generated using a keyword and used for encryption.
Theory:
The Playfair cipher is a digraph substitution cipher which means it encrypts pairs of letters rather than single letters. It's a comparatively simple polygraphic cipher that is encrypted by using a 5x5 matrix of letters.

The Playfair cipher requires a keyword which fills the 5x5 matrix with unique letters. The letters 'I' and 'J' are usually combined to fit within the 25 cells and then break the plaintext message into digraphs which is pair of letters.
If a digraph has repeating letters like "SS" an 'X' need to insert between them.
If the message has an odd number of letters, add an 'X' at the end.

For each pair of letters if f both letters are in the same row, replace them with the letters immediately to their right then if both letters are in the same column, replace them with the letters immediately below them.
If the letters are in different rows and columns that means in a form of rectangle then replace each letter with the letter in the same row but at the other corner of the rectangle.
Example:
Using the keyword "SHAMA," we construct the 5x5 matrix:

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

Let's encrypt the plaintext message "HELLO" following the Playfair rules:
At first divide into pairs, split "HELLO" into digraphs lik thia HE, LX, LO
As that "LL" is a repeated letter pair, so we insert an "X" between them, making it "LX."
Then encrypt each pair using the Playfair rules:

Pair 1: HE

H is in row 0, column 1; E is in row 1, column 2.
Since they are in different rows and columns, swap columns to form "AD."
Pair 2: LX
L is in row 2, column 2; X is in row 4, column 2.
Since they are in the same column, shift down by one: L → R, X → Y. This gives "RY."
Pair 3: LO
L is in row 2, column 2; O is in row 2, column 4.
Since they are in the same row, shift right by one: L → N, O → I. This gives "NI."
Combining the results, he ciphertext is ADRYNI.

Algorithm:

1.Generate a 5x5 Key Matrix:
2.Create the matrix with the keyword first including the keyword first, then fill in the remaining letters of the alphabet.
3.Divide the plaintext into pairs of letters and add filler 'X' for repeated letters or incomplete pairs.
4.For each pair of letters, use the Playfair cipher rules to find the corresponding ciphertext letters.

Code (Python):

```
def generate_playfair_matrix(keyword):
    keyword = "".join(sorted(set(keyword), key=keyword.index))  # Remove duplicates, preserve order
    matrix = ["" for _ in range(5)]
    used_chars = set(keyword.replace("J", "I"))
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  # Exclude J

    # Fill matrix with keyword
    row, col = 0, 0
    for char in keyword:
        matrix[row] += char
        col += 1
        if col == 5:
            row, col = row + 1, 0

    # Fill remaining spaces in matrix with unused characters
    for char in alphabet:
        if char not in used_chars:
            matrix[row] += char
            col += 1
            if col == 5:
                row, col = row + 1, 0

    return matrix

def find_position(matrix, char):
    for row in range(5):
        if char in matrix[row]:
            return row, matrix[row].index(char)
    return None

def encrypt_digraph(digraph, matrix):
    r1, c1 = find_position(matrix, digraph[0])
    r2, c2 = find_position(matrix, digraph[1])

    if r1 == r2:  # Same row
        return matrix[r1][(c1 + 1) % 5] + matrix[r2][(c2 + 1) % 5]
    elif c1 == c2:  # Same column
        return matrix[(r1 + 1) % 5][c1] + matrix[(r2 + 1) % 5][c2]
    else:  # Rectangle rule
        return matrix[r1][c2] + matrix[r2][c1]

def prepare_text(text):
    text = text.upper().replace("J", "I")
    prepared_text = ""
    i = 0

    while i < len(text):
        prepared_text += text[i]
        if i + 1 < len(text) and text[i] == text[i + 1]:
            prepared_text += "X"  # Insert X if double letters
        elif i + 1 < len(text):
            prepared_text += text[i + 1]
            i += 1
        i += 1

    if len(prepared_text) % 2 != 0:
        prepared_text += "X"  # Add X if odd-length

    return prepared_text

def playfair_encrypt(plaintext, keyword):
    matrix = generate_playfair_matrix(keyword)
    plaintext = prepare_text(plaintext)
    ciphertext = ""

    for i in range(0, len(plaintext), 2):
        ciphertext += encrypt_digraph(plaintext[i:i+2], matrix)

    return ciphertext
plaintext = "HELLO"
keyword = "SHAMA"
ciphertext = playfair_encrypt(plaintext, keyword)
print("Ciphertext:", ciphertext)
```

Input: Plain text: HELLO

Key word: SHAMA

Output: Ciphertext: ADRANI

Experiment No: 06

Experiment Name: Write a program to implement decryption using Playfair cipher.

Objective:

1. To know about the reverse mechanism of the Playfair cipher for secure message decryption.
2.To decrypt a given ciphertext using a keyword and a5x5 matrix made by it and validate the result.

Theory:

The Playfair cipher is a digraph substitution cipher that encrypts the plain text by dividing it into pairs of letters. Decryption means reversing the encryption rules used with the 5x5 matrix created from a keyword. For successful decryption it is important to know the same keyword used for encryption and follow the Playfair cipher decryption rules.

A 5x5 grid is generated using a keyword same as the encryption process. This matrix remains the same for both encryption and decryption. The keyword is followed by the remaining letters of the alphabet and I/J stands in the same cell.

At first the ciphertext needs to divide into pairs of letters. Then for each pair of letters, If the two letters are in the same row they are replaced with the letters to their immediate left and also wrapped around if needed). If the two letters are in the same column, replace them with the letters immediately above them and if the two letters form a rectangle, they are replaced with the letter in the same row but at the opposite corner of the rectangle. This is how the plain text will be found.

Example:

Keyword: "SHAMA"

Ciphertext: "ADRANI"

Break into pairs: AD RA NI

Decrypt using the matrix:

AD -> HE (rectangle rule)

RA -> LX (rectangle rule)

NI -> LO (rectangle rule)

Plaintext: "HELXLO" (which translates to "HELLO"

Algorithm:

1.Generate a 5x5 Key Matrix and use the keyword to generate the matrix for decryption.

2.Prepare the Ciphertext and divide it into pairs of letters.

3.Decrypt the Ciphertext using decryption rules of the Playfair cipher to get back the plaintext.

Code (Python):

```python
def generate_key_matrix(keyword):
    # Remove duplicates and convert to uppercase
    result = []
    for char in keyword.upper():
        if char not in result and char != 'J':  # J is replaced by I
            result.append(char)

    # Fill remaining alphabet (excluding 'J')
    for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':
        if char not in result:
            result.append(char)

    # Create 5x5 matrix
    matrix = [result[i:i + 5] for i in range(0, 25, 5)]
    return matrix

def find_position(matrix, char):
    for row in range(5):
        for col in range(5):
            if matrix[row][col] == char:
                return row, col
    return None

def decrypt_pair(matrix, char1, char2):
    row1, col1 = find_position(matrix, char1)
    row2, col2 = find_position(matrix, char2)

    if row1 == row2:  # Same row, shift left
        return matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]
    elif col1 == col2:  # Same column, shift up
        return matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]
    else:  # Rectangle case
        return matrix[row1][col2] + matrix[row2][col1]

def playfair_decrypt(ciphertext, keyword):
    matrix = generate_key_matrix(keyword)

    plaintext = ""
    for i in range(0, len(ciphertext), 2):
        char1 = ciphertext[i]
        char2 = ciphertext[i + 1]
        plaintext += decrypt_pair(matrix, char1, char2)

    return plaintext
```

```
# Example usage:
keyword = "SHAMA"
ciphertext = "ADRANI"
plaintext = playfair_decrypt(ciphertext, keyword)
print(f"Decrypted message: {plaintext}")
```

Input: Cipher Text: ADRANI
Key word: SHAMA
Output: Plain Text: HELXLO

Experiment No: 07

Experiment Name: Write a program to implement encryption using Poly-Alphabetic cipher.

Objective:

1.To understand the process of how multiple shifts are used in the Poly-Alphabetic cipher for stronger encryption.

2.To encrypt a message using a keyword and apply shifts accordingly and understand about how it secure for encryption.

Theory:

The Poly-Alphabetic cipher is an encryption technique where multiple substitution alphabets are used to encrypt the plaintext. This makes it more secure than the simple Caesar cipher, as here same letter can be encrypted differently depending on its position in the text. Vigenere cipher is one of the commonly used Poly alphabetic cipher.

At first a keyword is chosen and then repeated to match the length of the plaintext. Each letter in the plaintext is shifted by the corresponding letter in the keyword, where each letter in the keyword represents a shift value. Then each letter of the plaintext is shifted forward in the alphabet by the position of the corresponding keyword letter. This method uses modulo 26 arithmetic to wrap around the alphabet if the shift exceeds 'Z'.

$$C_i = (P_i + K_{i \bmod m}) \bmod 26$$

Example:

Keyword: "MIMROTA"

Plaintext: "SHAMA"

Encryption Steps:

Repeat the keyword to match the length of the plaintext: "MIMRO"

Then encrypt each letter:

S (18) + M (12) = E (using mod 26: (18 + 12) % 26 = 4)

H (7) + I (8) = P (using mod 26: (7 + 8) % 26 = 15)

A (0) + M (12) = M

M (12) + R (17) = D (using mod 26: (12 + 17) % 26 = 3)

A (0) + O (14) = O

Ciphertext: "EPMDO"

Algorithm:

1. At first take a Keyword and repeat it until it matches the length of the plaintext.

2.Convert each letter of the plaintext and the keyword into their numerical positions (A = 0, B = 1, ..., Z = 25).

3.For each letter in the plaintext shift it forward by the value of the corresponding letter in the keyword.
4. Apply modulo 26 arithmetic to ensure the shift wraps around if it exceeds 'Z'
5. Then convert the resulting numbers back to letters and form the ciphertext.

Code (Python):

```
def encrypt_poly_alphabetic(plaintext, keyword):
    # Convert both the plaintext and the keyword to uppercase
    plaintext = plaintext.upper()
    keyword = keyword.upper()

    # Create a list to store the ciphertext
    ciphertext = []

    # Repeat the keyword until it matches the length of the plaintext
    repeated_keyword = (keyword * (len(plaintext) // len(keyword) + 1))[:len(plaintext)]

    # Encrypt each letter
    for p_char, k_char in zip(plaintext, repeated_keyword):
        # Convert characters to their corresponding alphabet positions (A=0, B=1, ..., Z=25)
        p_value = ord(p_char) - ord('A')
        k_value = ord(k_char) - ord('A')

        # Shift the plaintext letter by the value of the corresponding keyword letter
        cipher_value = (p_value + k_value) % 26

        # Convert back to a letter and add to the ciphertext
        ciphertext.append(chr(cipher_value + ord('A')))

    # Join the list of ciphertext characters into a single string
    return ''.join(ciphertext)

# Example usage:
keyword = "MIMROTA"
plaintext = "SHAMA"
ciphertext = encrypt_poly_alphabetic(plaintext, keyword)
print(f"Encrypted message: {ciphertext}")
```

Input: Plain text: SHAMA
Key word: MIMROTA
Output: Cipher Text: EPMDO

Experiment No: 08

Experiment Name: Write a program to implement decryption using Poly-Alphabetic cipher.

Objective:

1. To understand how to reverse the multiple shifts that has been applied during encryption.

2.To decrypt a given ciphertext using the same keyword by which encryption has done

Theory:

The Poly-Alphabetic cipher uses multiple shifting alphabets for encryption and in description those shifts are reversed. Vigenère cipher is the most common form of Poly alphabetic cipher. In decryption, the same keyword that was used during encryption is used but instead of adding the shifts they are just subtracted to get the original plaintext.

$$P_i = (C_i - K_{i \bmod m}) \bmod 26$$

Here P represents Plain text and C represents Cipher text.

The same keyword used during encryption is used here and repeated to match the length of the ciphertext. For each letter in the ciphertext the corresponding keyword letter's shift value is subtracted. Both the ciphertext and keyword letters need to convert into their numerical positions. Then subtract the index of the keyword letter from the index of the ciphertext letter and mod 26 to wrap around if needed. After that the plain text letter is found out.

Example:

Keyword: "MIMROTA"

Ciphertext: "EPMDO"

Repeat the keyword to match the length of the ciphertext: "MIMRO"

Decrypt each letter:

E (4) - M (12) = S (using mod 26: (4 - 12 + 26) % 26 = 18)

P (15) - I (8) = H (using mod 26: (15 - 8) % 26 = 7)

M (12) - M (12) = A

D (3) - R (17) = M (using mod 26: (3 - 17 + 26) % 26 = 12)

O (14) - O (14) = A

Plaintext: "SHAMA"

Algorithm:

1. Generate Repeated Keyword and repeat the keyword until it matches the length of the ciphertext.

2.Decrypt Each Letter and convert the letters in the ciphertext and keyword to their numerical positions (A = 0, B = 1, ..., Z = 25).
3.For each letter in the ciphertext subtract the value of the corresponding keyword letter. Use modulo 26 arithmetic to handle negative results and wrap around the alphabet.

Code (Python):

```python
def decrypt_poly_alphabetic(ciphertext, keyword):
    # Convert both the ciphertext and the keyword to uppercase
    ciphertext = ciphertext.upper()
    keyword = keyword.upper()

    # Create a list to store the plaintext
    plaintext = []

    # Repeat the keyword until it matches the length of the ciphertext
    repeated_keyword = (keyword * (len(ciphertext) // len(keyword) + 1))[:len(ciphertext)]

    # Decrypt each letter
    for c_char, k_char in zip(ciphertext, repeated_keyword):
        # Convert characters to their corresponding alphabet positions (A=0, B=1, ..., Z=25)
        c_value = ord(c_char) - ord('A')
        k_value = ord(k_char) - ord('A')

        # Reverse the shift (subtract the keyword value)
        plain_value = (c_value - k_value + 26) % 26

        # Convert back to a letter and add to the plaintext
        plaintext.append(chr(plain_value + ord('A')))

    # Join the list of plaintext characters into a single string
    return ''.join(plaintext)

# Example usage:
keyword = "MIMROTA"
ciphertext = "EPMDO"
plaintext = decrypt_poly_alphabetic(ciphertext, keyword)
print(f"Decrypted message: {plaintext}")
```

Input: Key word: EPMDO
Cipher Text: MIMROTA
Output: Plain Text: SHAMA

Experiment No: 09

Experiment Name: Write a program to implement encryption using Vernam cipher.

Objective:

1. To understand the concept of the one-time pad, which Vernam cipher is based on.
2. To encrypt a message using a binary key that is as long as the plaintext.

Theory:

The Vernam cipher which is also known as the one-time pad, is a type of symmetric encryption. It is considered theoretically unbreakable if used correctly. The key used for encryption is as long as the plaintext and consists of random characters. Each character of the plaintext is XORed with the corresponding character of the key to generate the ciphertext.

$$Ci = Pi \oplus K$$

Here C is the cipher text and P is the plain text

The key used for encryption is a random sequence of characters which must be the same length as the plaintext. Each character in the plaintext is XORed with the corresponding character from the key. In binary terms, XOR (exclusive OR) results in a 1 when the two bits differ, and 0 when they are the same.

Convert both the plaintext and the key to their binary representations.Perform XOR between the binary values of the plaintext and the key.

Convert the result back to characters to form the ciphertext.

Example:

Plaintext: "SHAMA"

Key: "XMCKL" (randomly generated)

Encryption Steps:

Convert each character of the plaintext and the key to their binary equivalents.

XOR the binary values for each character:

S (binary: 01010011) XOR X (binary: 01011000) = K (binary: 00001011)

H (binary: 01001000) XOR M (binary: 01001101) = E (binary: 00000101)

A (binary: 01000001) XOR C (binary: 01000011) = B (binary: 00000010)

M (binary: 01001101) XOR K (binary: 01001011) = A (binary: 00000110)

A (binary: 01000001) XOR L (binary: 01001100) = M (binary: 00001101)

Ciphertext: "KEBAM"

Algorithm:

1. Generate a Random Key and the key must be the same length as the plaintext.
2. Convert both the plaintext and the key into their binary equivalents.

XOR Operation:

3. XOR each binary value from the plaintext with the corresponding binary value from the key.
4. Convert the XORed binary results back to characters to form the ciphertext.

Code (Python):

```
import random

alphabet = "abcdefghijklmnopqrstuvwxyz".upper()
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def generate_key(length):
    key = ""
    for i in range(length):
        key += chr(random.randint(65, 90))  # ASCII codes for A-Z
    return key


def encrypt(plaintext, key):
    ciphertext = ""
    cipherCode = []
    for i in range(len(plaintext)):
        xor = mp[plaintext[i]] ^ mp[key[i]]
        cipherCode.append(xor)
        ciphertext += mp2[(mp['A'] + xor) % 26]

    return ciphertext, cipherCode


def decrypt(cipherCode, key):
    plaintext = ""
    for i in range(len(cipherCode)):
        xor = cipherCode[i] ^ mp[key[i]]
        plaintext += mp2[xor % 26]
    return plaintext


plaintext = "SHAMA"
plaintext = plaintext.upper()
key = generate_key(len(plaintext))
ciphertext, cipherCode = encrypt(plaintext, key)
print("Ciphertext:", ciphertext)
```

Input: Plain text: Shama
Output: Cipher Text: QAABH

Experiment No: 10
Experiment Name: Write a program to implement decryption using Vernam cipher.
Objective:
1.To understand how the XOR operation is used to reverse the encryption and retrieve the original plaintext.
2.To decrypt a given ciphertext using the same key that was used during encryption.
Theory:
The Vernam cipher (or one-time pad) uses a random key that is as long as the plaintext to encrypt a message by applying the XOR (exclusive OR) operation. For decryption, the same XOR operation is used. Since XOR is its own inverse, applying XOR between the ciphertext and the key will retrieve the original plaintext.
The same random key used during encryption is required for decryption. The key must be as long as the cyphertext decryption, XOR the ciphertext with the same key. Since XORing the same bits twice returns the original bits, this will retrieve the original plaintext. Convert the ciphertext and key to their binary representations. Perform XOR between the binary values of the ciphertext and the key and convert the result back to characters to retrieve the plaintext.
Example:
Ciphertext: "KEBAM"
Key: "XMCKL" (same key used during encryption)
Decryption Steps:
Convert each character of the ciphertext and the key to their binary equivalents.
XOR the binary values for each character:
K (binary: 01001011) XOR X (binary: 01011000) = S (binary: 01010011)
E (binary: 01000101) XOR M (binary: 01001101) = H (binary: 01001000)
B (binary: 01000010) XOR C (binary: 01000011) = A (binary: 01000001)
A (binary: 01000001) XOR K (binary: 01001011) = M (binary: 01001101)
M (binary: 01001101) XOR L (binary: 01001100) = A (binary: 01000001)
Plaintext: "SHAMA"

Algorithm:
1.Take a randomly generated key and the key must be the same one used during encryption and must match the length of the ciphertext.
2.Convert both the ciphertext and the key into their binary equivalents.

3.XOR each binary value from the ciphertext with the corresponding binary value from the key.
4. Convert Binary Back to Characters:

Code (Python):

```
def decrypt_vernam_cipher(ciphertext, key):
  # Convert both the ciphertext and the key to uppercase
  ciphertext = ciphertext.upper()
  key = key.upper()

  # Create a list to store the plaintext
  plaintext = []

  # Decrypt each letter by XORing the binary values
  for c_char, k_char in zip(ciphertext, key):
    # Convert characters to their corresponding ASCII values
    c_value = ord(c_char)
    k_value = ord(k_char)

    # XOR the ASCII values
    plain_value = c_value ^ k_value

    # Convert back to a character and add to the plaintext
    plaintext.append(chr(plain_value))

  # Join the list of plaintext characters into a single string
  return ''.join(plaintext)

# Example usage:
ciphertext = "KEBAM"
key = "XMCKL"  # Same key used in encryption
plaintext = decrypt_vernam_cipher(ciphertext, key)

print(f"Ciphertext: {ciphertext}")
print(f"Key: {key}")
print(f"Decrypted message: {plaintext}")
```

Input:

Ciphertext: QAABH

Output:

Decrypted text: SHAMA