Pabna University of Science and Technology

Faculty of **Engineering and Technology**

Department of **Information and Communication Engineering**

Lab Report

Course Code: ICE-4108

Course Title: Cryptography and Computer Security Sessional

Submitted By:

Name: Md. Sha Alam

Roll No: 190610

Session: 2018-2019

4th Year 1st Semester

Department of ICE

PUST

Submitted To:

Tarun Debnath

Assistant Professor

Department of ICE

PUST

Submit Date : 27-02-2024

Signature

INDEX

Experiment No: 01

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

Objectives:

1. Understand Cryptography Concepts: Gain a practical understanding of basic cryptography principles, including symmetric encryption and substitution ciphers like the Caesar cipher.
2. Implement Encryption Algorithms: Learn how to implement a simple encryption algorithm in a programming language, which involves manipulating strings and characters.
3. Implement Decryption Algorithms: Understand the process of decrypting encrypted data, which involves reversing the encryption process.

Theory:

Caesar cipher is simplest form of substitution cipher. The Caesar cipher involves replacing each letter of the alphabet with the letter standing three places further down the alphabet. For example:

plain: meet me after the toga party
cipher: PHHW PH DIWHU WKH WRJD SDUWB

The transformation can be defined by listing all possibilities as follows:

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

Then the algorithm can be expressed as follows. For each plaintext letter, substitute the ciphertext letter:

$$C = E(3, p) = (p + 3) \bmod 26$$

A shift may be of any amount, so that the general Caesar algorithm is

$$C = E(k, p) = (p + k) \bmod 26$$

Where k takes on a value in the range 1 to 25. The decryption algorithm is simply

$$p = D(k, C) = (C - k) \bmod 26$$

Example: Let us consider an example:

Plaintext: cryptography and computer network

a	b	c	d	e	f	g	h	i	j	k	l	m
D	E	F	G	H	I	J	K	L	M	N	O	P
n	o	p	q	r	s	t	u	v	w	x	y	z
Q	R	S	T	U	V	W	X	Y	Z	A	B	C

In this example, the shift k=3

So, the encrypted ciphertext will be: FUBSFWJUDSKB DQG FRPSXWHU QHWZRUN

Ciphertext: FUBSFWJUDSKBDQGFRPSXWHUQHWZRUN

Algorithm for Caesar Cipher:

Input:

- A String of lower-case letters, called Text.
- An Integer between 0-25 denoting the required shift.

Procedure:

- Traverse the given text one character at a time.
- For each character, transform the given character as per the rule, depending on whether we're encrypting or decrypting the text.
- Return the new string generated.

Source 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 = input()
shift = int(input())
ciphertext = caesar_encrypt(plaintext, shift)
print(f"Caesar Cipher Encryption: {ciphertext}")
plaintext = caesar_decrypt(ciphertext, shift)
print(f"Caesar Cipher Decryption: {plaintext}")
```

Output:

Caesar Cipher Encryption: phhw ph diwhu wkh wrjd sduwb
Caesar Cipher Decryption: meet me after the toga party

Experiment No: 02

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

Objectives:

1. Understanding of Cryptography Concepts: Gain practical knowledge of encryption techniques by implementing a basic symmetric encryption algorithm.
2. Exploration of Substitution Ciphers: Understand the concept of substitution ciphers, particularly the Mono-Alphabetic cipher, which involves substituting each letter of the plaintext with a corresponding letter from the cipher alphabet.

Theory:

Monoalphabetic cipher is a substitution cipher in which for a given key, the cipher alphabet for each plain alphabet is fixed throughout the encryption process. For example, if ‘A’ is encrypted as ‘D’, for any number of occurrences in that plaintext, ‘A’ will always get encrypted to ‘D’.
In this technique, any plaintext letter can be substituted with any of the ciphertext letters, which means no ciphertext letter can not be repeated. Such an approach is referred to as monoalphabetic substitution cipher because a single cipher alphabet is used per message.
Example: Let us consider an example:

Plaintext: cryptography and computer network

a	b	c	d	e	f	g	h	i	j	k	l	m
D	J	U	B	N	G	S	K	P	F	W	Q	L
n	o	p	q	r	s	t	u	v	w	x	y	z
I	C	O	H	V	T	A	Y	X	M	Z	R	E

So, the encrypted ciphertext will be: UVROASVDOKR DIB UCLOYANV INAMCVW

Ciphertext: UVROASVDOKRDIBUCLOYANVINAMCVW

Monoalphabetic cipher is easy to break because it reflects the frequency data of the original alphabet. A countermeasure is to provide multiple substitutes, known as homophones, for a single letter.
For example, the letter e could be assigned a number of different cipher symbols, such as 16, 74, 35, and 21, with each homophone assigned to a letter in rotation or randomly. If the number of symbols assigned to each letter is proportional to the relative frequency of that letter, then single-letter frequency information is completely obliterated
However, even with homophones, each element of plaintext affects only one element of ciphertext, and multiple-letter patterns (e.g., diagram frequencies) still survive in the ciphertext, making cryptanalysis relatively straightforward.

Source 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 = input()

    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()
```

Output:

Plain text: ShaAlam
Encrypted message: LIQQSQD
Decrypted message: shaalam

Experiment No: 03

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

Objectives:

1. Algorithmic Understanding: Gain a deep understanding of the Playfair cipher algorithm, including how to create and use the key matrix and the rules for encrypting and decrypting plaintext. This objective focuses on mastering the core principles of the cipher and its implementation.
2. Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information, including understanding the strengths and weaknesses of the Playfair cipher compared to other classical ciphers. This objective emphasizes understanding the security implications of using the Playfair cipher in practical scenarios.

Theory:

The Playfair algorithm is based on the use of a 5 × 5 matrix of letters constructed using a keyword. Here is an example, solved by Lord Peter Wimsey in Dorothy Sayers's Have His Carcase:

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

In this case, the keyword is monarchy. The matrix is constructed by filling in the letters of the keyword (minus duplicates) from left to right and from top to bottom, and then filling in the remainder of the matrix with the remaining letters in alphabetic order. The letters I and J count as one letter. Plaintext is encrypted two letters at a time, according to the following rules:

1. Repeating plaintext letters that are in the same pair are separated with a filler letter, such as x, so that balloon would be treated as ba lx lo on.
2. Two plaintext letters that fall in the same row of the matrix are each replaced by the letter to the right, with the first element of the row circularly following the last. For example, ar is encrypted as RM.
3. Two plaintext letters that fall in the same column are each replaced by the letter beneath, with the top element of the column circularly following the last. For example, mu is encrypted as CM.
4. Otherwise, each plaintext letter in a pair is replaced by the letter that lies in its own row and the column occupied by the other plaintext letter. Thus, hs becomes BP and ea becomes IM (or JM, as the encipherer wishes).

The Playfair Cipher Algorithm: The Algorithm mainly consist of three steps:

- Convert plaintext into digraphs (i.e., into pair of two letters)
- Generate a Cipher Key Matrix
- Encrypt plaintext using Cipher Key Matrix and get ciphertext.

Source Code:

```python
def playfair_cipher(plaintext, key, mode):
    # Define the alphabet, excluding 'j'
    alphabet = 'abcdefghiklmnopqrstuvwxyz'
    # Remove whitespace and 'j' from the key and convert to lowercase
    key = key.lower().replace(' ', '').replace('j', 'i')
    # Construct the key square
    key_square = ''
    for letter in key + alphabet:
        if letter not in key_square:
            key_square += letter
    # Split the plaintext into digraphs, padding with 'x' if necessary
    plaintext = plaintext.lower().replace(' ', '').replace('j', 'i')
    replaceplaintext = ''
    if mode == 'encrypt':
        it = 0
        while it < len(plaintext) - 1:
            if plaintext[it] == plaintext[it + 1]:
                replaceplaintext += plaintext[it]
                replaceplaintext += 'x'
                it += 1
            else:
                replaceplaintext += plaintext[it]
                replaceplaintext += plaintext[it + 1]
                it += 2
        replaceplaintext += plaintext[-1] if it < len(plaintext) else ''
        plaintext = replaceplaintext

    if len(plaintext) % 2 == 1:
        plaintext += 'x'
    digraphs = [plaintext[i:i + 2] for i in range(0, len(plaintext), 2)]

    # Define the encryption/decryption functions
    def encrypt(digraph):
        a, b = digraph
        row_a, col_a = divmod(key_square.index(a), 5)
        row_b, col_b = divmod(key_square.index(b), 5)
        if row_a == row_b:
            col_a = (col_a + 1) % 5
            col_b = (col_b + 1) % 5
        elif col_a == col_b:
            row_a = (row_a + 1) % 5
            row_b = (row_b + 1) % 5
        else:
            col_a, col_b = col_b, col_a
        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

    def decrypt(digraph):
        a, b = digraph
        row_a, col_a = divmod(key_square.index(a), 5)
        row_b, col_b = divmod(key_square.index(b), 5)
        if row_a == row_b:
            col_a = (col_a - 1) % 5
            col_b = (col_b - 1) % 5
        elif col_a == col_b:
            row_a = (row_a - 1) % 5
            row_b = (row_b - 1) % 5
        else:
            col_a, col_b = col_b, col_a
        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]
```

```python
    # Encrypt or decrypt the plaintext
    result = ''
    for digraph in digraphs:
        if mode == 'encrypt':
            result += encrypt(digraph)
        elif mode == 'decrypt':
            result += decrypt(digraph)

    # Return the result
    return result


# Example usage
plaintext = input()
key = input()
ciphertext = playfair_cipher(plaintext, key, 'encrypt')
print('Cipher Text:', ciphertext)
decrypted_text = playfair_cipher(ciphertext, key, 'decrypt')
print('Decrypted Text:', decrypted_text)  # (Note: 'x' is added as padding)

```

Output:

Cipher Text: tgeyioco
Decrypted Text: shaxalam

Experiment No: 04

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

Objectives:

1. Algorithmic Understanding: Gain a deep understanding of the Hill cipher algorithm, including matrix multiplication and modular arithmetic. Understand how the key matrix is used for encryption and decryption, and how it affects the security of the cipher.
2. Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information. Explore the strengths and weaknesses of the Hill cipher, particularly in terms of its susceptibility to attacks such as known-plaintext attacks and key size limitations.

Theory:

Hill cipher is a polygraphic substitution cipher based on linear algebra.Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible $n \times n$ matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible $n \times n$ matrices (modulo 26).

Source Code:

```python
import string
import numpy as np

alphabet = string.ascii_lowercase
letter_to_index = dict(zip(alphabet, range(len(alphabet))))
index_to_letter = dict(zip(range(len(alphabet)), alphabet))


def egcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        gcd, x, y = egcd(b % a, a)
        return gcd, y - (b // a) * x, x


def mod_inv(det, modulus):
    gcd, x, y = egcd(det, modulus)
    if gcd != 1:
        raise Exception("Matrix is not invertible.")
    return (x % modulus + modulus) % modulus


def matrix_mod_inv(matrix, modulus):
    det = int(np.round(np.linalg.det(matrix)))
    det_inv = mod_inv(det, modulus)
    matrix_modulus_inv = (
        det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus
    )
    return matrix_modulus_inv.astype(int)


def encrypt_decrypt(message, K):
    msg = ""
    message_in_numbers = [letter_to_index[letter] for letter in message]
    split_P = [
        message_in_numbers[i: i + len(K)]
        for i in range(0, len(message_in_numbers), len(K))
    ]
    for P in split_P:
        P = np.transpose(np.asarray(P))[:, np.newaxis]
        while P.shape[0] != len(K):
            P = np.append(P, letter_to_index[" "])[:, np.newaxis]
        numbers = np.dot(K, P) % len(alphabet)
        n = numbers.shape[0]
        for idx in range(n):
            number = int(numbers[idx, 0])
            msg += index_to_letter[number]
    return msg


message = input()
K = np.matrix([[3, 3], [2, 5]])
Kinv = matrix_mod_inv(K, len(alphabet))

encrypted_message = encrypt_decrypt(message, K)
decrypted_message = encrypt_decrypt(encrypted_message, Kinv)

print("Original message: " + message.upper())
print("Encrypted message: " + encrypted_message.upper())
print("Decrypted message: " + decrypted_message.upper())

```

Output:

Original message: IAMSHAALAM
Encrypted message: YQMKVOHDKI
Decrypted message: IAMSHAALAM

Experiment No: 05

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

Objectives:

1. Algorithmic Understanding: Develop a deep understanding of the Poly-Alphabetic cipher algorithm, particularly focusing on the concept of using multiple alphabets (key streams) to encrypt plaintext. Understand how to create and manage key streams, and how they are applied during encryption and decryption.
2. Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information. Explore the security properties of the Poly-Alphabetic cipher, including its resistance to frequency analysis attacks due to the variability introduced by multiple key streams. Understand its strengths and limitations compared to other classical ciphers.

Theory:

One way to improve on the simple monoalphabetic technique is to use different monoalphabetic substitutions as one proceeds through the plaintext message. The general name for this approach is polyalphabetic substitution cipher. All these techniques have the following features in common:

1. A set of related monoalphabetic substitution rules is used.

2. A key determines which particular rule is chosen for a given transformation.

Vigenère Cipher: The best known, and one of the simplest, polyalphabetic ciphers is the Vigenère cipher. In this scheme, the set of related monoalphabetic substitution rules consists of the 26 Caesar ciphers with shifts of 0 through 25. Each cipher is denoted by a key letter, which is the ciphertext letter that substitutes for the plaintext letter a. Thus, a Caesar cipher with a shift of 3 is denoted by the key value.

A general equation of the encryption process is

Ci = (pi + kimod m) mod 26

decryption is a generalization of Equation

pi = (Ci - kimod m) mod 26

Example: To encrypt a message, a key is needed that is as long as the message. Usually, the key is a repeating keyword. For example,

if the keyword is deceptive, the message “we are discovered save yourself” is encrypted as

key: deceptivedeceptivedeceptive

plaintext: wearediscoveredsaveyourself

ciphertext: ZICVTWQNGRZGVTWAVZHCQYGLMGJ

Expressed numerically, we have the following result

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**
0	1	2	3	4	5	6	7	8	9	10	11	12
n	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
13	14	15	16	17	18	19	20	21	22	23	24	25

key	3	4	2	4	15	19	8	21	4	3	4	2	4	15
Plaintext	22	4	0	17	4	3	8	18	2	14	21	4	17	4
Ciphertext	25	8	2	21	19	22	16	13	6	17	25	6	21	19

Key	19	8	21	4	3	4	2	4	15	19	8	21	4
Plaintext	3	18	0	21	4	14	20	20	17	18	4	11	5
Ciphertext	22	0	21	25	7	2	16	24	6	11	13	6	9

Source Code:

```
# Poly_alphabetic cipher

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


def generateKey(plainText, keyword):
    key = ''
    for i in range(len(plainText)):
        key += keyword[i % len(keyword)]
    return key


def cipherText(plainText, key):
    cipher_text = ""
    for i in range(len(plainText)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[plainText[i].upper()] + shift) % 26]
        cipher_text += newChar
    return cipher_text


def decrypt(cipher_text, key):
    plainText = ''
    for i in range(len(cipher_text)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[cipher_text[i].upper()] - shift + 26) % 26
        plainText += newChar
    return plainText


plainText = input()
keyword = input()
key = generateKey(plainText, keyword)
cipher_text = cipherText(plainText, key)
print("Ciphertext :", cipher_text)
print("Decrypted Text :", decrypt(cipher_text, key))
```

Output:
Ciphertext : AJEINEU
Decrypted Text : SHAALAM

Experiment No: 06

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

Objectives:

1. Algorithmic Understanding: Gain a deep understanding of the Vernam cipher algorithm, also known as the one-time pad, including how it uses a random or pseudorandom key stream to encrypt plaintext and decrypt ciphertext.
2. Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information. Explore the security properties of the Vernam cipher, particularly its perfect secrecy when used with a truly random key stream of equal length as the plaintext.

Theory:

The Vernam cipher, also known as the one-time pad, is a symmetric encryption algorithm that uses the principle of the XOR (exclusive OR) operation to combine a random or pseudorandom key stream with plaintext to produce ciphertext, and vice versa for decryption.

Encryption: To encrypt a message, the Vernam cipher requires a key stream of the same length as the plaintext. Each character in the plaintext is combined with the corresponding character in the key stream using the XOR operation. The result is the ciphertext.

Decryption: To decrypt the ciphertext, the same key stream used for encryption is XORed with the ciphertext. This operation retrieves the original plaintext.

Key Generation: The security of the Vernam cipher relies on the randomness and secrecy of the key stream. Ideally, the key stream should be generated using a truly random process and should only be known to the sender and the intended recipient.

Perfect Secrecy: The Vernam cipher provides perfect secrecy when used with a truly random key stream of equal length as the plaintext. This means that even with unlimited computational resources, an attacker cannot gain any information about the plaintext from the ciphertext.

Key Reuse: One of the critical requirements of the Vernam cipher is that the key stream must never be reused for encrypting more than one message. Reusing the key stream compromises the security of the cipher and can lead to the recovery of the plaintext through statistical analysis or brute force attacks.

By implementing the Vernam cipher, learners can deepen their understanding of encryption concepts, improve their programming skills, and explore the practical applications of encryption techniques in cybersecurity and information security.

Source 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 = input()
plaintext = plaintext.upper()
key = generate_key(len(plaintext))
ciphertext, cipherCode = encrypt(plaintext, key)
print("Ciphertext:", ciphertext)
decryptedtext = decrypt(cipherCode, key)
print("Decrypted text:", decryptedtext)

```

Output:

Ciphertext: ZKIUYPC
Decrypted text: SHAALAM

Experiment No: 07

Experiment Name: Write a program to implement encryption and decryption using Brute force attack cipher.

Objectives:

1. Understanding of Cryptanalysis: Gain practical knowledge of cryptanalysis techniques, particularly brute force attacks, which involve systematically trying all possible keys until the correct one is found.
2. Algorithmic Understanding: Understand the structure and characteristics of the cipher being targeted by the brute force attack. This includes understanding the encryption algorithm, the key space, and any known vulnerabilities or weaknesses that can be exploited.
3. Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information. Explore the limitations of brute force attacks, including their computational complexity and effectiveness against different types of ciphers.

Theory:

Understanding the Brute Force Attack:

A brute force attack is an exhaustive search method that systematically tries all possible combinations until the correct solution is found.
In the context of decryption, a brute force attack involves trying every possible key to decrypt the ciphertext until the original plaintext is recovered.
Brute force attacks are often used when no other attack method is feasible, such as when the encryption key is unknown or the encryption algorithm is resistant to other types of attacks.

Key Space:

The key space refers to the total number of possible keys that can be used in the encryption algorithm.
For symmetric encryption algorithms, such as Caesar cipher or Vigenère cipher, the key space represents all possible shifts or combinations of characters in the key.
The size of the key space determines the feasibility of a brute force attack. A larger key space makes brute force attacks more computationally expensive and time-consuming.

Implementation:

To implement a brute force attack for decryption, start by generating all possible keys within the key space.
For each generated key, decrypt the ciphertext using the encryption algorithm.
Compare the decrypted plaintext with known patterns or characteristics of the original plaintext.
If a match is found, the correct key has been discovered, and the decryption process can be stopped.
If no match is found after trying all possible keys, the ciphertext may be incorrectly encrypted, or the key space may be too large for a brute force attack to be practical.

Computational Complexity:

The computational complexity of a brute force attack depends on the size of the key space.
For encryption algorithms with small key spaces, such as Caesar cipher with a shift of 1, a brute force attack can be performed quickly.
However, for encryption algorithms with large key spaces, such as modern block ciphers like AES with a 128-bit key, brute force attacks are computationally infeasible due to the vast number of possible keys.
Limitations:

Brute force attacks are not always practical, especially for encryption algorithms with large key spaces.
They require significant computational resources and time to search through all possible keys.
Brute force attacks may also be ineffective against encryption algorithms with additional security features, such as key stretching or password hashing.
In summary, implementing encryption and decryption using a brute force attack involves systematically trying all possible keys until the correct one is found. The feasibility of a brute force attack depends on the size of the key space and the computational resources available.

Source Code:

```python
def brute_force_decrypt(ciphertext):
    for shift in range(26):
        decrypted_text = caesar_decrypt(ciphertext, shift)
        print(f"Shift {shift}: {decrypted_text}")


def brute_force_encrypt(plainText):
    for shift in range(26):
        encrypted_text = caesar_encrypt(plainText, shift)
        print(f"Shift {shift}: {encrypted_text}")


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


plantext=input()
print('Brute Force Encryption for Caesar Cipher:')
brute_force_encrypt(plantext)

ciphertext = "ifmmp"
print("\nBrute Force Decryption for Caesar Cipher:")
brute_force_decrypt(ciphertext)
```

Output:
Brute Force Encryption for Caesar Cipher:
Shift 0: meet me after the toga party
Shift 1: nffu nf bgufs uif uphb qbsuz
Shift 2: oggv og chvgt vjg vqic rctva
Shift 3: phhw ph diwhu wkh wrjd sduwb
Shift 4: qiix qi ejxiv xli xske tevxc
Shift 5: rjjy rj fkyjw ymj ytlf ufwyd

Shift 6: skkz sk glzkx znk zumg vgxze
Shift 7: tlla tl hmaly aol avnh whyaf
Shift 8: ummb um inbmz bpm bwoi xizbg
Shift 9: vnnc vn jocna cqn cxpj yjach
Shift 10: wood wo kpdob dro dyqk zkbdi
Shift 11: xppe xp lqepc esp ezrl alcej
Shift 12: yqqf yq mrfqd ftq fasm bmdfk
Shift 13: zrrg zr nsgre gur gbtn cnegl
Shift 14: assh as othsf hvs hcuo dofhm
Shift 15: btti bt puitg iwt idvp epgin
Shift 16: cuuj cu qvjuh jxu jewq fqhjo
Shift 17: dvvk dv rwkvi kyv kfxr grikp
Shift 18: ewwl ew sxlwj lzw lgys hsjlq
Shift 19: fxxm fx tymxk max mhzt itkmr
Shift 20: gyyn gy uznyl nby niau julns
Shift 21: hzzo hz vaozm ocz ojbv kvmot
Shift 22: iaap ia wbpan pda pkcw lwnpu
Shift 23: jbbq jb xcqbo qeb qldx mxoqv
Shift 24: kccr kc ydrcp rfc rmey nyprw
Shift 25: ldds ld zesdq sgd snfz ozqsx

Brute Force Decryption for Caesar Cipher:
Shift 0: nffu nf bgufs uif uphb qbsuz
Shift 1: meet me after the toga party
Shift 2: ldds ld zesdq sgd snfz ozqsx
Shift 3: kccr kc ydrcp rfc rmey nyprw
Shift 4: jbbq jb xcqbo qeb qldx mxoqv
Shift 5: iaap ia wbpan pda pkcw lwnpu
Shift 6: hzzo hz vaozm ocz ojbv kvmot
Shift 7: gyyn gy uznyl nby niau julns
Shift 8: fxxm fx tymxk max mhzt itkmr
Shift 9: ewwl ew sxlwj lzw lgys hsjlq
Shift 10: dvvk dv rwkvi kyv kfxr grikp
Shift 11: cuuj cu qvjuh jxu jewq fqhjo
Shift 12: btti bt puitg iwt idvp epgin
Shift 13: assh as othsf hvs hcuo dofhm
Shift 14: zrrg zr nsgre gur gbtn cnegl
Shift 15: yqqf yq mrfqd ftq fasm bmdfk
Shift 16: xppe xp lqepc esp ezrl alcej
Shift 17: wood wo kpdob dro dyqk zkbdi
Shift 18: vnnc vn jocna cqn cxpj yjach
Shift 19: ummb um inbmz bpm bwoi xizbg
Shift 20: tlla tl hmaly aol avnh whyaf
Shift 21: skkz sk glzkx znk zumg vgxze
Shift 22: rjjy rj fkyjw ymj ytlf ufwyd
Shift 23: qiix qi ejxiv xli xske tevxc
Shift 24: phhw ph diwhu wkh wrjd sduwb
Shift 25: oggv og chvgt vjg vqic rctva