**PANDIT DEENDAYAL ENERGY UNIVERSITY**

**SCHOOL OF TECHNOLOGY**

# Course: Information Security

**Course Code: 20CP304P**

**LAB MANUAL**

**B.Tech. (Computer Engineering)**

**Semester 5**

**Submitted To:**

Dr. Hargeet Kaur

**Submitted By:**

Aryan Randeriya

22BCP469D

G3

# INDEX

| S. No. | List of experiments | Date | Sign |
|---|---|---|---|
| **1** | Study and Implement a program for Caesar Cipher | | |
| **2** | Study and Implement a program for 5x5 Playfair Cipher to encrypt and decrypt the message. | | |
| **3** | Study and Implement a program for Rail Fence Cipher with columnar transposition | | |
| **4** | Study and implement a program for columnar Transposition Cipher | | |
| **5** | Study and implement a program for Vigenère Cipher | | |
| **6** | Study and Implement a program for n-gram Hill Cipher | | |
| **7** | Study and Use of RSA algorithm (encryption and decryption) | | |
| **8** | Study and implement a program of the Digital Signature with RSA algorithm (Reverse RSA) | | |
| **9** | Study and Use of Diffie-Hellman Key Exchange | | |
| **10** | Design cipher with detailed explanation. Sample as follows.<br>a) Write a program to encrypt the plaintext with the given key. E.g. plaintext GRONSFELD with the key 1234. Add 1 to G to get H (the letter 1 rank after G is H in the alphabet), then add 2 to C or E (the letter 2 ranks after C is E), and so on. Use smallest letter from plaintext as filler.<br>b) Encrypt the input words PLAINTEXT= RAG BABY to obtain CIPHERTEXT = SCJ DDFD | | |

## Experiment 4

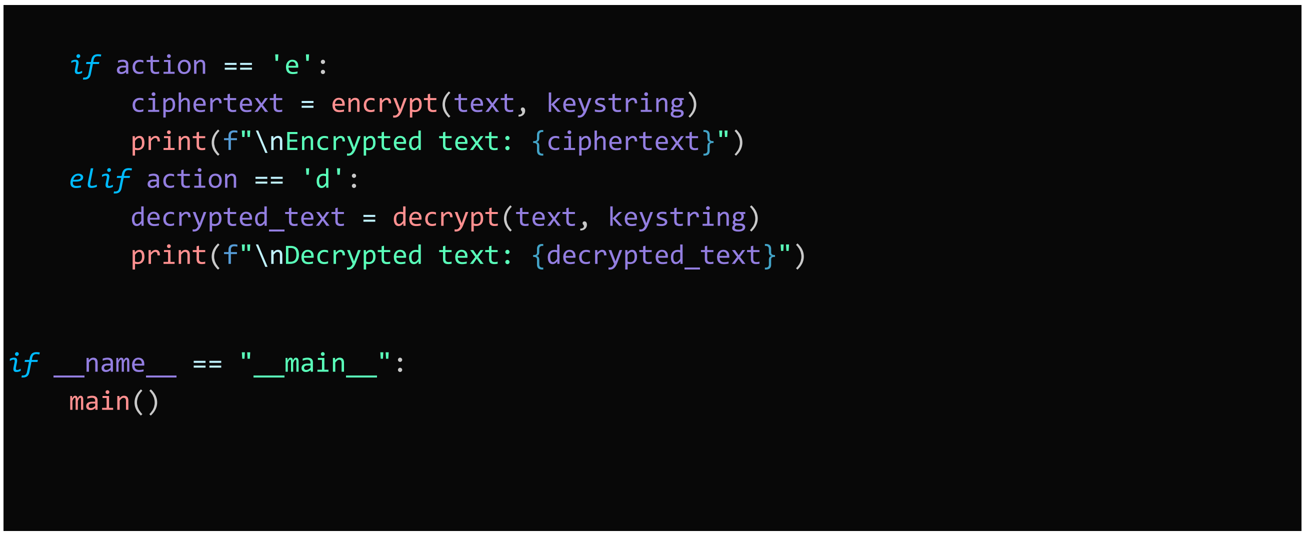
**AIM:** Study and Implement a program for Columnar Transposition Cipher

## Introduction:

The Columnar Transposition Cipher is a type of transposition cipher where the plaintext is written in rows into a matrix, and then read column-by-column based on a key that defines the column order. The key determines the order in which the columns are rearranged to form the ciphertext,

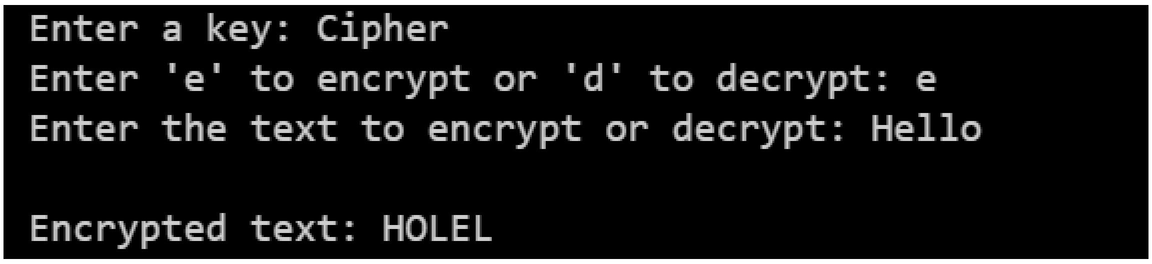
## Program:

```python
def encrypt(plaintext, key):
    # Remove spaces from the plaintext and convert to uppercase
    plaintext = plaintext.replace(' ', '').upper()

    # Determine the number of columns and rows
    num_cols = len(key)
    num_rows = len(plaintext) // num_cols
    if len(plaintext) % num_cols != 0:
        num_rows += 1

    # Create a matrix for the plaintext
    matrix = [['' for _ in range(num_cols)] for _ in range(num_rows)]

    # Fill the matrix with the plaintext
    for i, char in enumerate(plaintext):
        row = i // num_cols
        col = i % num_cols
        matrix[row][col] = char

    # Create a list of columns based on the key order
    col_order = sorted(range(len(key)), key=lambda k: key[k])

    # Read columns in order
    ciphertext = ''
    for col in col_order:
        for row in range(num_rows):
            if matrix[row][col] != '':
                ciphertext += matrix[row][col]

    return ciphertext


def decrypt(ciphertext, key):
    # Determine the number of columns and rows
    num_cols = len(key)
    num_rows = len(ciphertext) // num_cols
    if len(ciphertext) % num_cols != 0:
        num_rows += 1
    matrix = [['' for _ in range(num_cols)] for _ in range(num_rows)]

    # Create a list of columns based on the key order
    col_order = sorted(range(len(key)), key=lambda k: key[k])

    # Fill the matrix with the ciphertext
    index = 0
    for col in col_order:
        for row in range(num_rows):
            if index < len(ciphertext):
                matrix[row][col] = ciphertext[index]
                index += 1

    # Read the matrix in row-wise order
    plaintext = ''
    for row in range(num_rows):
        for col in range(num_cols):
            if matrix[row][col] != '':
                plaintext += matrix[row][col]

    return plaintext


def user_input() -> tuple:
    while True:
        try:
            # Prompt user for the key and convert it to an integer
            keystring: int = input("Enter a key: ")

            # Prompt user for the action ('e' for encrypt or 'd' for decrypt)
            action: str = input(
                "Enter 'e' to encrypt or 'd' to decrypt: ").lower()

            # Check if the action is valid
            if action not in ['e', 'd']:
                raise ValueError(
                    "Action must be 'e' for encryption or 'd' for decryption.")

            # Prompt user for the plaintext or ciphertext
            text: str = input(
                "Enter the text to encrypt or decrypt: ").replace(" ", "")

            return keystring, action, text
        except ValueError as e:
            # Display error message and prompt user to try again if input is
invalid
            print(f"Error: {e}")
            print("Please try again.")


def main():
    """Main function to handle user input and perform encryption or decryption."""
    keystring, action, text = user_input()
```

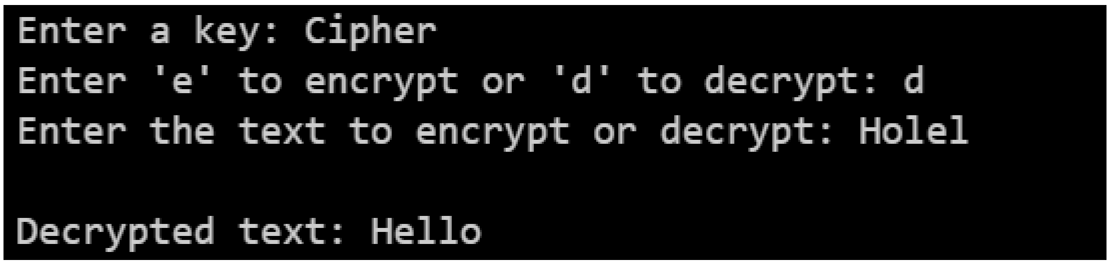


```python
    if action == 'e':
        ciphertext = encrypt(text, keystring)
        print(f"\nEncrypted text: {ciphertext}")
    elif action == 'd':
        decrypted_text = decrypt(text, keystring)
        print(f"\nDecrypted text: {decrypted_text}")


if __name__ == "__main__":
    main()
```

**Output:**



```
Enter a key: Cipher
Enter 'e' to encrypt or 'd' to decrypt: e
Enter the text to encrypt or decrypt: Hello

Encrypted text: HOLEL
```



```
Enter a key: Cipher
Enter 'e' to encrypt or 'd' to decrypt: d
Enter the text to encrypt or decrypt: Holel

Decrypted text: Hello
```

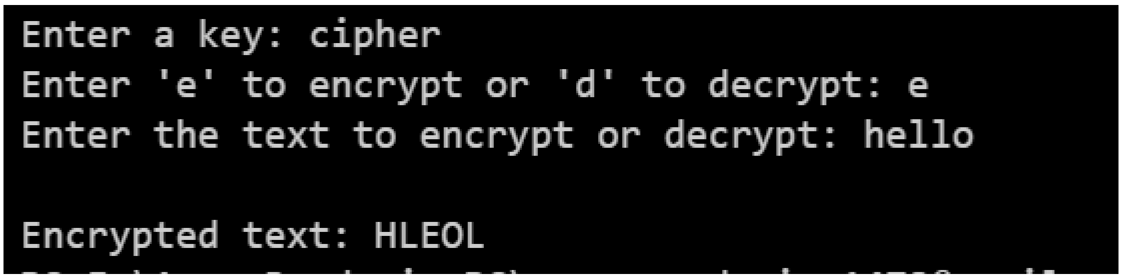
## Revised Approach

The revised Columnar Transposition Cipher involves performing double transposition, where the plaintext is first transposed based on a key and then re-transposed using either the same or a different key. This adds an additional layer of complexity to the cipher, making it harder to crack. Optionally, the process can be repeated for multiple rounds of transposition, further enhancing the security by increasing the level of scrambling of the text.

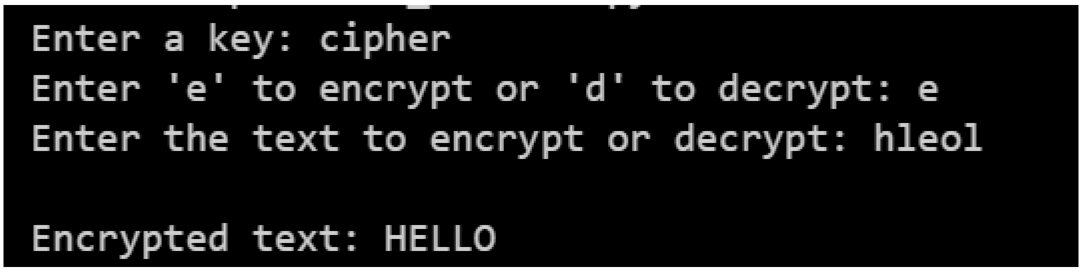
**Code:**

```python
def encrypt(plaintext, key):
    # Remove spaces from the plaintext and convert to uppercase
    plaintext = plaintext.replace(' ', '').upper()

    # Determine the number of columns and rows
    num_cols = len(key)
    num_rows = len(plaintext) // num_cols
    if len(plaintext) % num_cols != 0:
        num_rows += 1

    # Create a matrix for the plaintext
    matrix = [['' for _ in range(num_cols)] for _ in range(num_rows)]

    # Fill the matrix with the plaintext
    for i, char in enumerate(plaintext):
        row = i // num_cols
        col = i % num_cols
        matrix[row][col] = char

    # Create a list of columns based on the key order
    col_order = sorted(range(len(key)), key=lambda k: key[k])

    # Read columns in order
    ciphertext = ''
    for col in col_order:
        for row in range(num_rows):
            if matrix[row][col] != '':
                ciphertext += matrix[row][col]

    return ciphertext


def decrypt(ciphertext, key):
    # Determine the number of columns and rows
    num_cols = len(key)
    num_rows = len(ciphertext) // num_cols
    if len(ciphertext) % num_cols != 0:
        num_rows += 1

    # Create a matrix for the ciphertext
    matrix = [['' for _ in range(num_cols)] for _ in range(num_rows)]

    # Create a list of columns based on the key order
    col_order = sorted(range(len(key)), key=lambda k: key[k])
    # Fill the matrix with the ciphertext
    index = 0
    for col in col_order:
        for row in range(num_rows):
            if index < len(ciphertext):
                matrix[row][col] = ciphertext[index]
                index += 1

    # Read the matrix in row-wise order
    plaintext = ''
    for row in range(num_rows):
        for col in range(num_cols):
            if matrix[row][col] != '':
                plaintext += matrix[row][col]

    return plaintext


def user_input() -> tuple:
    while True:
        try:
            # Prompt user for the key and convert it to an integer
            keystring: int = input("Enter a key: ")

            # Prompt user for the action ('e' for encrypt or 'd' for decrypt)
            action: str = input(
                "Enter 'e' to encrypt or 'd' to decrypt: ").lower()

            # Check if the action is valid
            if action not in ['e', 'd']:
                raise ValueError(
                    "Action must be 'e' for encryption or 'd' for decryption.")

            # Prompt user for the plaintext or ciphertext
            text: str = input(
                "Enter the text to encrypt or decrypt: ").replace(" ", "")

            return keystring, action, text
        except ValueError as e:
            # Display error message and prompt user to try again if input is
invalid
            print(f"Error: {e}")
            print("Please try again.")


def main():
    """Main function to handle user input and perform encryption or decryption."""
    keystring, action, text = user_input()

    if action == 'e':
        ciphertext = encrypt(text, keystring)
        ciphertext = encrypt(ciphertext, keystring)
        print(f"\nEncrypted text: {ciphertext}")
    elif action == 'd':
        decrypted_text = decrypt(text, keystring)
        decrypted_text = decrypt(decrypted_text, keystring)
        print(f"\nDecrypted text: {decrypted_text}")


if __name__ == "__main__":
    main()
```

## Output:



```
Enter a key: cipher
Enter 'e' to encrypt or 'd' to decrypt: e
Enter the text to encrypt or decrypt: hello

Encrypted text: HLEOL
```



```
Enter a key: cipher
Enter 'e' to encrypt or 'd' to decrypt: e
Enter the text to encrypt or decrypt: hleol

Encrypted text: HELLO
```

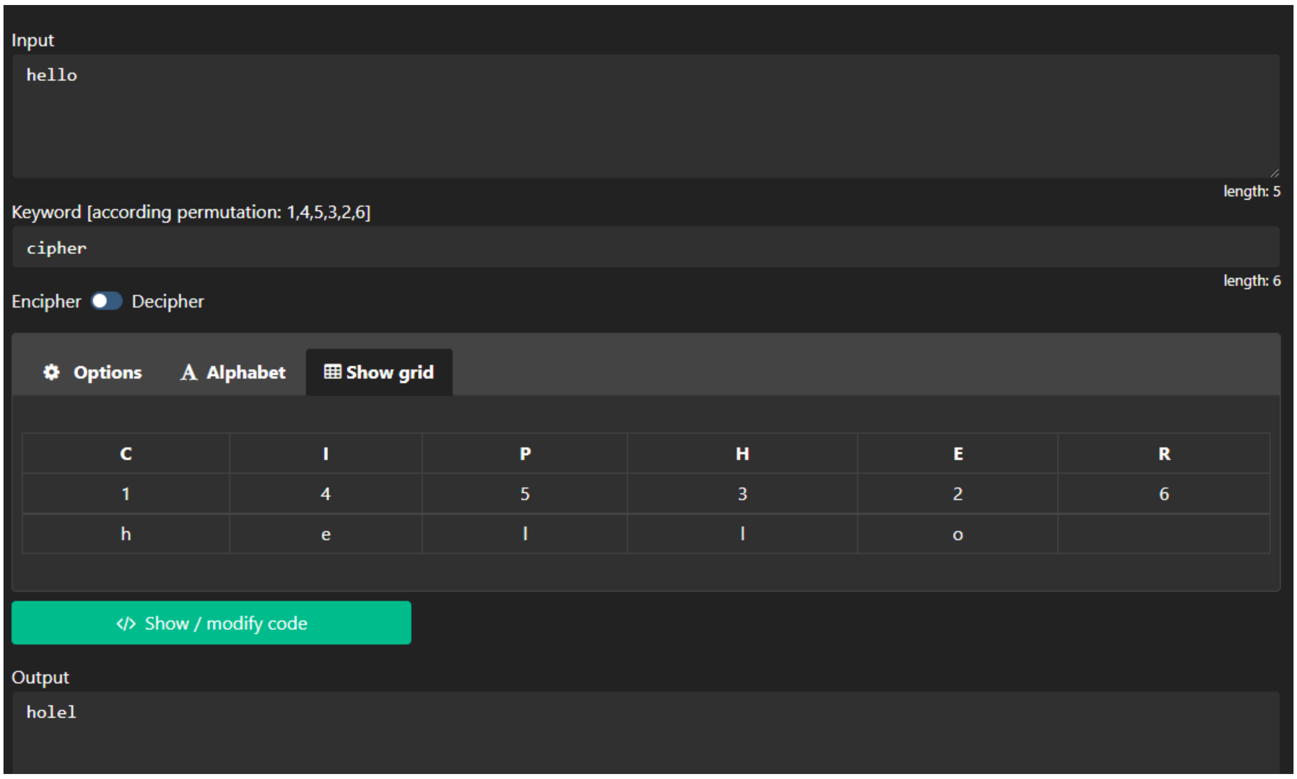
**Comparative Analysis of original and revised approach:**
The original Columnar Transposition Cipher performs a single rearrangement of the plaintext based on a key, offering basic security through column reordering. However, it is vulnerable to frequency analysis and can be cracked relatively easily. In contrast, the revised approach introduces **double transposition**, where the text is transposed twice, significantly increasing the complexity and resistance to cryptanalysis.

**Crypt Analysis:**

**Anagramming:** Since the cipher only changes the order of the letters without altering them, one approach is anagramming—finding sequences in the ciphertext that make sense when rearranged. Attackers may test different key lengths and column permutations to reorganize the letters into a readable form.

**Brute Force Attack:**
For relatively short keys, brute force (trying every possible columnar arrangement) is feasible. If the key length is known or guessed, all possible permutations of the key can be tested to see which one produces readable text.

## CrypTool Output:


Input

```
hello
```

length: 5

Keyword [according permutation: 1,4,5,3,2,6]

```
cipher
```

length: 6

Encipher Decipher

| C | I | P | H | E | R |
|---|---|---|---|---|---|
| 1 | 4 | 5 | 3 | 2 | 6 |
| h | e | l | l | o | |

Output

```
holel
```

## Conclusion:

The Columnar Transposition Cipher offers a basic level of security by rearranging characters according to a key, but it remains vulnerable to cryptanalytic techniques like frequency analysis and anagramming. The revised approach, involving double transposition and multiple rounds of scrambling, significantly enhances the cipher's resilience against attacks by adding complexity and making pattern detection more difficult. While the original version is suitable for simple encryption, the revised method is a more secure and robust choice for protecting sensitive information.

## References:

- https://www.dcode.fr/columnar-transposition-cipher
- https://www.nku.edu/~christensen/1402%20Columnar%20transposition.pdf