

Name: Shrey Pendurkar

Class: D15C

Batch: C

Roll No: 64

CNS - Experiment 4

Aim: Design and Implement a product cipher using Substitution ciphers and Transposition Cipher.

Theory:

A product cipher is a type of encryption scheme that combines multiple encryption techniques, such as substitution and transposition ciphers, in order to enhance security. By applying multiple layers of encryption, product ciphers aim to provide a higher level of complexity and make it more difficult for attackers to break the cipher.

Substitution Cipher:

The substitution cipher is a method where each letter of the plaintext is replaced with another letter or symbol according to a fixed rule or key. For example, a simple substitution cipher may replace each letter with the letter that appears three positions after it in the alphabet.

- a. Substitution Encryption: Apply a substitution cipher to the plaintext to produce a modified plaintext (substituted characters).
- b. Transposition Encryption: Rearrange the substituted characters from the previous step using a transposition cipher to create the ciphertext.

Transposition Cipher:

The transposition cipher is a method where the letters of the plaintext are rearranged or permuted based on a fixed rule or key. For example, a simple transposition cipher may rearrange the letters in the plaintext in a specific pattern, such as reading them in columns instead of rows.

- a. Transposition Decryption: Reverse the transposition cipher to rearrange the ciphertext into the substituted characters.
- b. Substitution Decryption: Apply the inverse of the substitution cipher to obtain the original plaintext.

Review Questions

1. What is a Product Cipher?

A **product cipher** is a method of encryption that combines two or more simpler encryption transformations to build a stronger cipher. The purpose is to leverage the strengths of each individual method (like substitution and permutation) so that the resulting cipher resists various cryptanalysis techniques more effectively than the components alone.

Historically, product ciphers were introduced to overcome the limitations of classical ciphers such as simple substitution or transposition ciphers, which are vulnerable to frequency analysis or pattern detection.

2. Basic Structure and Components of a Product Cipher

A typical product cipher has the following components:

- **Substitution Layer (S-boxes):**

- This replaces each small piece of the plaintext (such as a nibble or byte) with another piece based on a predefined substitution table.
- This introduces **confusion**, meaning the relationship between the plaintext and ciphertext becomes complex and non-linear.

- **Permutation Layer (P-boxes):**

- This rearranges bits or groups of bits across the entire block of data.
- This provides **diffusion**, which means that the influence of one plaintext bit is spread over many ciphertext bits, hiding statistical patterns.

- **Key Schedule:**

- Generates round keys for each iteration from the main encryption key.
 - **Rounds:**
 - The substitution and permutation steps are repeated multiple times (rounds) to compound security.
-

3. Encryption Process in Detail

Step-by-step example:

Suppose we have a 16-bit plaintext block.

- **Round 1:**
 1. **Substitution:**

Break the 16 bits into four 4-bit sections. Each 4-bit block passes through an S-box, substituting it with another 4-bit value (e.g., 1100 → 0101).
 2. **Permutation:**

Rearrange all 16 bits according to a fixed permutation pattern (e.g., bit 1 goes to bit 5, bit 2 goes to bit 12, etc.).
 3. **Key mixing:**

XOR the output with a round key derived from the main key.
- **Repeat:**

This sequence (substitution → permutation → key mixing) repeats for several rounds (say 8 rounds).
- **Output:**

After the final round, the resulting bits form the ciphertext.

This layered approach ensures that by the last round, changing one bit in the plaintext affects many bits in the ciphertext, and the relationship between input and output is highly non-linear.

4. How Product Ciphers Enhance Security

- **Confusion:** The substitution step hides the relationship between plaintext bits and ciphertext bits. It prevents attackers from making simple connections.
 - **Diffusion:** The permutation step ensures that changing a single bit in the plaintext affects many ciphertext bits (avalanche effect), making statistical analysis more difficult.
 - **Multiple rounds:** Each round increases complexity exponentially. An attacker cannot easily reverse the process without the key.
 - **Combining different operations:** Even if an attacker finds a weakness in the substitution step, the permutation step adds complexity, and vice versa.
-

5. Dependence on Strength and Secrecy

- If an S-box is poorly designed (e.g., linear or predictable), attackers can exploit that to mount attacks like linear or differential cryptanalysis.
 - If permutation patterns are weak or predictable, diffusion is limited.
 - If keys are leaked or poorly managed, no amount of substitution or permutation will secure the message.
 - Therefore, **each component must be carefully designed and the keys kept secret** for the product cipher to be secure.
-

6. Limitations and When Product Ciphers May Be Less Effective

- **If component ciphers are weak:** For example, if the substitution or permutation is trivial or has patterns, attackers can break the cipher.
 - **Performance concerns:** Multiple rounds of substitution and permutation can slow down encryption, especially in constrained devices.
 - **Modern ciphers like AES:** These are already well-studied product ciphers with carefully designed components. Using custom product ciphers instead of AES or other modern ciphers can be risky and unnecessary.
- Applications with different needs:** Sometimes, stream ciphers or public-key cryptography might be preferred over block-based product ciphers.

Code:

```
import math

plaintext = input("Enter the plaintext to be encrypted: ").replace(" ", "").upper()

def letter_to_num(c):
    return "ABCDEFGHIJKLMNOPQRSTUVWXYZ".index(c)

def num_to_letter(n):
    return "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[n]

def caesar_cipher_encrypt(a):
    b = letter_to_num(a)
    return num_to_letter((b + 3) % 26)

def caesar_cipher_decrypt(a):
    b = letter_to_num(a)
    return num_to_letter((b - 3) % 26)

# Encryption

pt_list = list(plaintext)
par_encryp_text = [caesar_cipher_encrypt(char) for char in pt_list]
```

```

print("After Caesar cipher:", "".join(par_encrypt_text))

# Calculate rows and columns for transposition
n_chars = len(par_encrypt_text)
n_cols = math.ceil(math.sqrt(n_chars))
n_rows = math.ceil(n_chars / n_cols)

# Create matrix and fill column-wise
matrix = [["" for _ in range(n_cols)] for _ in range(n_rows)]

index = 0
for col in range(n_cols):
    for row in range(n_rows):
        if index < n_chars:
            matrix[row][col] = par_encrypt_text[index]
            index += 1
        else:
            matrix[row][col] = "" # padding

print("\nMatrix:")
for row in matrix:
    print(" ".join(char if char != "" else "_" for char in row))

# Read row-wise to get final encrypted text
final_text = ""
for row in range(n_rows):
    for col in range(n_cols):
        if matrix[row][col] != "":
            final_text += matrix[row][col]

print("\nFinal Encrypted Text:", final_text)

# Decryption

def decrypt(final_text):
    n_chars = len(final_text)
    n_cols = math.ceil(math.sqrt(n_chars))
    n_rows = math.ceil(n_chars / n_cols)

```

```

# Calculate number of full columns
full_cols = n_chars % n_rows
if full_cols == 0:
    full_cols = n_cols # all columns full

# Number of characters in each column during encryption:
# columns before full_cols have n_rows chars,
# others have n_rows - 1 chars (because of padding)
col_lengths = [n_rows if c < full_cols else n_rows - 1 for c in
range(n_cols)]

# Now reconstruct matrix column-wise using col_lengths
matrix = [["" for _ in range(n_cols)] for _ in range(n_rows)]
index = 0
for col in range(n_cols):
    for row in range(col_lengths[col]):
        if index < n_chars:
            matrix[row][col] = final_text[index]
            index += 1

# Read row-wise to get Caesar cipher text
caesar_text = ""
for row in range(n_rows):
    for col in range(n_cols):
        if matrix[row][col] != "":
            caesar_text += matrix[row][col]

# Reverse Caesar cipher
decrypted_text = ""
for char in caesar_text:
    decrypted_text += caesar_cipher_decrypt(char)

return decrypted_text

decrypted_text = decrypt(final_text)
print("\nDecrypted Text:", decrypted_text)

```

Output:

```
Enter the plaintext to be encrypted: mynameissshreypendurkar
After Caesar cipher: PBQDPHLVVKUHBSHQGXUNDU
```

Matrix:

P	H	U	Q	D
B	L	H	G	U
Q	V	B	X	_
D	V	S	U	_
P	K	H	N	_

Final Encrypted Text: PHUQDBLHGUVQVBXDVSUPKHN

Decrypted Text: MYNAMEISSSHREYPENDURKAR