**SUBMITTED BY**

**AAKRITI MEHROTRA**

**22BCE1954**

**LAB1**

**CAESAR CIPHER**

**SERVER CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

#define SHIFT 5 // Shift value (k)

void encrypt(char \*message, int shift)

{

for (int i = 0; message[i] != '\0'; i++)

{

if (message[i] >= 'a' && message[i] <= 'z')

{

message[i] = ((message[i] - 'a' + shift) % 26) + 'a';

}

}

}

int main()

{

int server\_fd, new\_socket;

struct sockaddr\_in address;

int opt = 1;

int addrlen = sizeof(address);

char buffer[BUFFER\_SIZE] = {0};

// Creating socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0)

{

perror("Socket failed");

exit(EXIT\_FAILURE);

}

// Forcefully attaching socket to the port 8080

if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt)))

{

perror("Setsockopt failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Binding socket

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0)

{

perror("Bind failed");

exit(EXIT\_FAILURE);

}

// Listening for connections

if (listen(server\_fd, 3) < 0)

{

perror("Listen failed");

exit(EXIT\_FAILURE);

}

printf("Server is listening on port %d\n", PORT);

// Accepting connection

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t \*)&addrlen)) < 0)

{

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// Encrypt and send a message

char message[] = "hello";

printf("Original message: %s\n", message);

encrypt(message, SHIFT); // Encrypt the message

send(new\_socket, message, strlen(message), 0);

printf("Encrypted message sent: %s\n", message);

close(new\_socket);

close(server\_fd);

return 0;

}

**CLIENT CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

#define SHIFT 5 // Same shift value (k)

void decrypt(char \*message, int shift)

{

for (int i = 0; message[i] != '\0'; i++)

{

if (message[i] >= 'a' && message[i] <= 'z')

{

message[i] = ((message[i] - 'a' - shift + 26) % 26) + 'a'; // +26 ensures positive result

}

}

}

int main()

{

int sock = 0;

struct sockaddr\_in serv\_addr;

char buffer[BUFFER\_SIZE] = {0};

// Creating socket

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

perror("Socket creation error");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Converting IP address to binary

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0)

{

perror("Invalid address or address not supported");

return -1;

}

// Connecting to server

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)

{

perror("Connection failed");

return -1;

}

// Receiving encrypted message

int valread = read(sock, buffer, BUFFER\_SIZE);

buffer[valread] = '\0';

printf("Encrypted message received: %s\n", buffer);

// Decrypting message

decrypt(buffer, SHIFT);

printf("Decrypted message: %s\n", buffer);

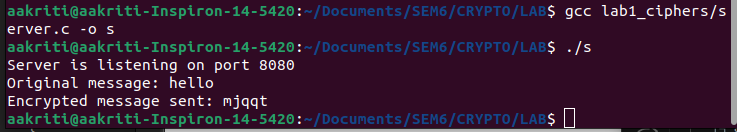
close(sock);

return 0;

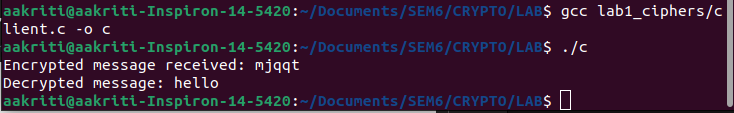
}

**OUTPUT:-**

**SERVER**

****

**CLIENT**

****

**SVIGNERE CIPHER**

**SERVER CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

#define SHIFT 5 // Shift value (k)

// Vigenère cipher encryption function

void encrypt(char \*message, const char \*key)

{

int messageLen = strlen(message);

int keyLen = strlen(key);

char extendedKey[messageLen + 1];

// Repeat the key to match the size of the plaintext

for (int i = 0; i < messageLen; i++)

{

extendedKey[i] = key[i % keyLen];

}

extendedKey[messageLen] = '\0';

// Perform the Vigenère cipher encryption

for (int i = 0; i < messageLen; i++)

{

if (message[i] >= 'a' && message[i] <= 'z') // Assuming lowercase letters

{

int pi = message[i] - 'a';

int ki = extendedKey[i] - 'a';

message[i] = ((pi + ki) % 26) + 'a';

}

}

}

int main()

{

int server\_fd, new\_socket;

struct sockaddr\_in address;

int opt = 1;

int addrlen = sizeof(address);

char buffer[BUFFER\_SIZE] = {0};

// Creating socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0)

{

perror("Socket failed");

exit(EXIT\_FAILURE);

}

// Forcefully attaching socket to the port 8080

if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt)))

{

perror("Setsockopt failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Binding socket

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0)

{

perror("Bind failed");

exit(EXIT\_FAILURE);

}

// Listening for connections

if (listen(server\_fd, 3) < 0)

{

perror("Listen failed");

exit(EXIT\_FAILURE);

}

printf("Server is listening on port %d\n", PORT);

// Accepting connection

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t \*)&addrlen)) < 0)

{

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// Encrypt and send a message

char message[] = "wearediscovered";

const char key[] = "deceptive";

printf("Original message: %s\n", message);

encrypt(message, key); // Encrypt the message using Vigenère cipher

send(new\_socket, message, strlen(message), 0);

printf("Encrypted message sent: %s\n", message);

close(new\_socket);

close(server\_fd);

return 0;

}

**CLIENT CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

// Vigenère cipher decryption function

void decrypt(char \*message, const char \*key)

{

int messageLen = strlen(message);

int keyLen = strlen(key);

char extendedKey[messageLen + 1];

// Repeat the key to match the size of the ciphertext

for (int i = 0; i < messageLen; i++)

{

extendedKey[i] = key[i % keyLen];

}

extendedKey[messageLen] = '\0';

// Perform the Vigenère cipher decryption

for (int i = 0; i < messageLen; i++)

{

if (message[i] >= 'a' && message[i] <= 'z') // Assuming lowercase letters

{

int ci = message[i] - 'a';

int ki = extendedKey[i] - 'a';

message[i] = ((ci - ki + 26) % 26) + 'a';

}

}

}

int main()

{

int sock = 0;

struct sockaddr\_in serv\_addr;

char buffer[BUFFER\_SIZE] = {0};

// Creating socket

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

perror("Socket creation error");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Converting IP address to binary

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0)

{

perror("Invalid address or address not supported");

return -1;

}

// Connecting to server

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)

{

perror("Connection failed");

return -1;

}

// Receiving encrypted message

int valread = read(sock, buffer, BUFFER\_SIZE);

buffer[valread] = '\0';

printf("Encrypted message received: %s\n", buffer);

// Decrypting message

const char key[] = "deceptive"; // Same key used for encryption

decrypt(buffer, key);

printf("Decrypted message: %s\n", buffer);

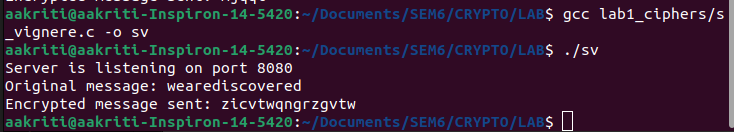
close(sock);

return 0;

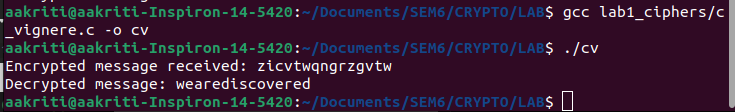
}

**OUTPUT:-**

**SERVER**

****

**CLIENT**

****

**VERNAM CIPHER**

**SERVER CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

// Vernam cipher encryption function

void encrypt(char \*plaintext, char \*key, char \*ciphertext)

{

int len = strlen(plaintext);

for (int i = 0; i < len; i++)

{

ciphertext[i] = ((plaintext[i] ^ key[i]) % 26) + 'A';

}

ciphertext[len] = '\0';

}

int main()

{

int server\_fd, new\_socket;

struct sockaddr\_in address;

int opt = 1;

int addrlen = sizeof(address);

// Creating socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0)

{

perror("Socket failed");

exit(EXIT\_FAILURE);

}

// Forcefully attaching socket to the port 8080

if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt)))

{

perror("Setsockopt failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Binding socket

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0)

{

perror("Bind failed");

exit(EXIT\_FAILURE);

}

// Listening for connections

if (listen(server\_fd, 3) < 0)

{

perror("Listen failed");

exit(EXIT\_FAILURE);

}

printf("Server is listening on port %d\n", PORT);

// Accepting connection

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t \*)&addrlen)) < 0)

{

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// Prepare plaintext and key

char plaintext[] = "HAI";

char key[] = "SAY"; // Key must have the same size as plaintext

char ciphertext[BUFFER\_SIZE];

// Encrypt the plaintext

encrypt(plaintext, key, ciphertext);

printf("Plaintext: %s\n", plaintext);

printf("Key: %s\n", key);

printf("Ciphertext: %s\n", ciphertext);

// Send the ciphertext and key to the client

send(new\_socket, ciphertext, strlen(ciphertext), 0);

send(new\_socket, key, strlen(key), 0);

printf("Encrypted message and key sent to client.\n");

close(new\_socket);

close(server\_fd);

return 0;

}

**CLIENT CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

// Vernam cipher decryption function

void decrypt(char \*ciphertext, char \*key, char \*plaintext)

{

int len = strlen(ciphertext);

for (int i = 0; i < len; i++)

{

plaintext[i] = ((ciphertext[i] ^ key[i]) % 26) + 'A';

}

plaintext[len] = '\0';

}

int main()

{

int sock = 0;

struct sockaddr\_in serv\_addr;

char ciphertext[BUFFER\_SIZE] = {0};

char key[BUFFER\_SIZE] = {0};

char plaintext[BUFFER\_SIZE];

// Creating socket

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

perror("Socket creation error");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Converting IP address to binary

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0)

{

perror("Invalid address or address not supported");

return -1;

}

// Connecting to server

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)

{

perror("Connection failed");

return -1;

}

// Receiving ciphertext and key

int valread = read(sock, ciphertext, BUFFER\_SIZE);

ciphertext[valread] = '\0';

valread = read(sock, key, BUFFER\_SIZE);

key[valread] = '\0';

printf("Ciphertext received: %s\n", ciphertext);

printf("Key received: %s\n", key);

// Decrypt the ciphertext

decrypt(ciphertext, key, plaintext);

printf("Decrypted message: %s\n", plaintext);

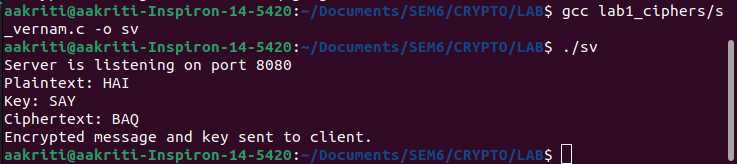
close(sock);

return 0;

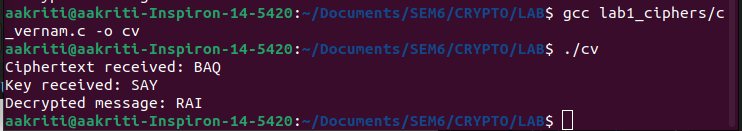
}

**OUTPUT:-**

**SERVER**

****

**CLIENT**

****

**LAB2**

**HILL CIPHER**

**SERVER CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

#define MATRIX\_SIZE 2

// Matrix multiplication for encryption

void matrix\_multiply(int plaintext[MATRIX\_SIZE][MATRIX\_SIZE], int key[MATRIX\_SIZE][MATRIX\_SIZE], int ciphertext[MATRIX\_SIZE][MATRIX\_SIZE])

{

for (int i = 0; i < MATRIX\_SIZE; i++)

{

for (int j = 0; j < MATRIX\_SIZE; j++)

{

ciphertext[i][j] = 0;

for (int k = 0; k < MATRIX\_SIZE; k++)

{

ciphertext[i][j] += plaintext[i][k] \* key[k][j];

}

ciphertext[i][j] %= 26; // Modulo 26

}

}

}

// Convert text to a matrix representation

void text\_to\_matrix(char \*text, int matrix[MATRIX\_SIZE][MATRIX\_SIZE])

{

for (int i = 0; i < MATRIX\_SIZE; i++)

{

for (int j = 0; j < MATRIX\_SIZE; j++)

{

matrix[i][j] = text[i \* MATRIX\_SIZE + j] - 'A';

}

}

}

// Convert matrix to text representation

void matrix\_to\_text(int matrix[MATRIX\_SIZE][MATRIX\_SIZE], char \*text)

{

for (int i = 0; i < MATRIX\_SIZE; i++)

{

for (int j = 0; j < MATRIX\_SIZE; j++)

{

text[i \* MATRIX\_SIZE + j] = matrix[i][j] + 'A';

}

}

text[MATRIX\_SIZE \* MATRIX\_SIZE] = '\0';

}

int main()

{

int server\_fd, new\_socket;

struct sockaddr\_in address;

int opt = 1;

int addrlen = sizeof(address);

// Predefined key matrix

int key[MATRIX\_SIZE][MATRIX\_SIZE] = {{9, 4}, {5, 7}};

// Creating socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0)

{

perror("Socket failed");

exit(EXIT\_FAILURE);

}

// Forcefully attaching socket to the port 8080

if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt)))

{

perror("Setsockopt failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Binding socket

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0)

{

perror("Bind failed");

exit(EXIT\_FAILURE);

}

// Listening for connections

if (listen(server\_fd, 3) < 0)

{

perror("Listen failed");

exit(EXIT\_FAILURE);

}

printf("Server is listening on port %d\n", PORT);

// Accepting connection

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t \*)&addrlen)) < 0)

{

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// Encrypt plaintext using Hill cipher

char plaintext[] = "MEET";

int plaintext\_matrix[MATRIX\_SIZE][MATRIX\_SIZE];

int ciphertext\_matrix[MATRIX\_SIZE][MATRIX\_SIZE];

char ciphertext[BUFFER\_SIZE];

text\_to\_matrix(plaintext, plaintext\_matrix);

matrix\_multiply(plaintext\_matrix, key, ciphertext\_matrix);

matrix\_to\_text(ciphertext\_matrix, ciphertext);

printf("Plaintext: %s\n", plaintext);

printf("Ciphertext: %s\n", ciphertext);

// Send ciphertext to client

send(new\_socket, ciphertext, strlen(ciphertext), 0);

printf("Ciphertext sent to client.\n");

close(new\_socket);

close(server\_fd);

return 0;

}

**CLIENT CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

#define MATRIX\_SIZE 2

// Matrix multiplication for decryption

void matrix\_multiply(int ciphertext[MATRIX\_SIZE][MATRIX\_SIZE], int key\_inverse[MATRIX\_SIZE][MATRIX\_SIZE], int plaintext[MATRIX\_SIZE][MATRIX\_SIZE])

{

for (int i = 0; i < MATRIX\_SIZE; i++)

{

for (int j = 0; j < MATRIX\_SIZE; j++)

{

plaintext[i][j] = 0;

for (int k = 0; k < MATRIX\_SIZE; k++)

{

plaintext[i][j] += ciphertext[i][k] \* key\_inverse[k][j];

}

plaintext[i][j] = (plaintext[i][j] % 26 + 26) % 26; // Modulo 26 to handle negatives

}

}

}

// Convert text to a matrix representation

void text\_to\_matrix(char \*text, int matrix[MATRIX\_SIZE][MATRIX\_SIZE])

{

for (int i = 0; i < MATRIX\_SIZE; i++)

{

for (int j = 0; j < MATRIX\_SIZE; j++)

{

matrix[i][j] = text[i \* MATRIX\_SIZE + j] - 'A';

}

}

}

// Convert matrix to text representation

void matrix\_to\_text(int matrix[MATRIX\_SIZE][MATRIX\_SIZE], char \*text)

{

for (int i = 0; i < MATRIX\_SIZE; i++)

{

for (int j = 0; j < MATRIX\_SIZE; j++)

{

text[i \* MATRIX\_SIZE + j] = matrix[i][j] + 'A';

}

}

text[MATRIX\_SIZE \* MATRIX\_SIZE] = '\0';

}

int main()

{

int sock = 0;

struct sockaddr\_in serv\_addr;

char ciphertext[BUFFER\_SIZE];

char plaintext[BUFFER\_SIZE];

// Predefined inverse key matrix

int key\_inverse[MATRIX\_SIZE][MATRIX\_SIZE] = {{5, 12}, {15, 25}};

// Creating socket

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

perror("Socket creation error");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Converting IP address to binary

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0)

{

perror("Invalid address or address not supported");

return -1;

}

// Connecting to server

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)

{

perror("Connection failed");

return -1;

}

// Receiving ciphertext

int valread = read(sock, ciphertext, BUFFER\_SIZE);

ciphertext[valread] = '\0';

printf("Ciphertext received: %s\n", ciphertext);

// Decrypt ciphertext using Hill cipher

int ciphertext\_matrix[MATRIX\_SIZE][MATRIX\_SIZE];

int plaintext\_matrix[MATRIX\_SIZE][MATRIX\_SIZE];

text\_to\_matrix(ciphertext, ciphertext\_matrix);

matrix\_multiply(ciphertext\_matrix, key\_inverse, plaintext\_matrix);

matrix\_to\_text(plaintext\_matrix, plaintext);

printf("Decrypted message: %s\n", plaintext);

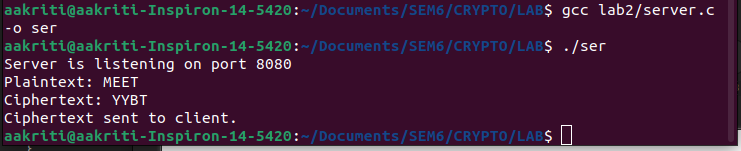
close(sock);

return 0;

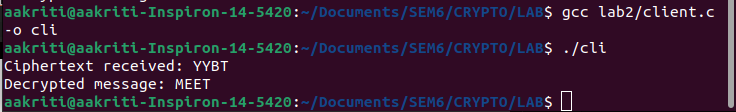
}

**OUTPUT:-**

**SERVER**



**CLIENT**



**LAB - 3**

**TRANSPOSITION – RAILFENCE CIPHER**

**SERVER SIDE CODE**

import socket

def encrypt\_rail\_fence(text, rails):

rail = [['\n' for \_ in range(len(text))] for \_ in range(rails)]

dir\_down = False

row, col = 0, 0

for i in range(len(text)):

if (row == 0) or (row == rails - 1):

dir\_down = not dir\_down

rail[row][col] = text[i]

col += 1

row += 1 if dir\_down else -1

result = []

for i in range(rails):

for j in range(len(text)):

if rail[i][j] != '\n':

result.append(rail[i][j])

return "".join(result)

# Server setup

HOST = '127.0.0.1' # Localhost

PORT = 65432 # Port to listen on

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as server\_socket:

server\_socket.bind((HOST, PORT))

server\_socket.listen()

print(f"Server listening on {HOST}:{PORT}...")

conn, addr = server\_socket.accept()

with conn:

print(f"Connected by {addr}")

data = conn.recv(1024).decode()

if data:

plaintext, rails = data.split('|')

rails = int(rails)

encrypted\_text = encrypt\_rail\_fence(plaintext, rails)

conn.sendall(encrypted\_text.encode())

**CLIENT SIDE CODE**

import socket

def decrypt\_rail\_fence(cipher, rails):

rail = [['\n' for \_ in range(len(cipher))] for \_ in range(rails)]

dir\_down = None

row, col = 0, 0

for i in range(len(cipher)):

if (row == 0) or (row == rails - 1):

dir\_down = not dir\_down

rail[row][col] = '\*'

col += 1

row += 1 if dir\_down else -1

index = 0

for i in range(rails):

for j in range(len(cipher)):

if (rail[i][j] == '\*') and (index < len(cipher)):

rail[i][j] = cipher[index]

index += 1

result = []

row, col = 0, 0

for i in range(len(cipher)):

if (row == 0):

dir\_down = True

if (row == rails - 1):

dir\_down = False

if rail[row][col] != '\*':

result.append(rail[row][col])

col += 1

row += 1 if dir\_down else -1

return "".join(result)

# Client setup

HOST = '127.0.0.1' # Server's hostname or IP address

PORT = 65432 # Port used by the server

plaintext = "HELLO I AM WORKING WELL"

rails = 3

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as client\_socket:

client\_socket.connect((HOST, PORT))

client\_socket.sendall(f"{plaintext}|{rails}".encode())

encrypted\_text = client\_socket.recv(1024).decode()

print(f"Encrypted Text from Server: {encrypted\_text}")

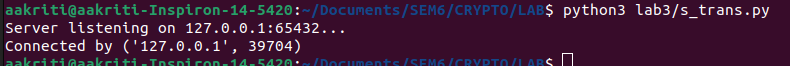
# Decrypt the encrypted text locally

decrypted\_text = decrypt\_rail\_fence(encrypted\_text, rails)

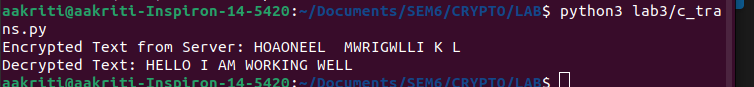
print(f"Decrypted Text: {decrypted\_text}")

**OUTPUT-**

SERVER SIDE



CLIENT SIDE



**PLAYFAIR ENCRYPTION**

**SERVER SIDE CODE**

import socket

import numpy as np

def to\_lower\_case(text):

return text.lower()

def remove\_spaces(text):

return text.replace(" ", "")

def generate\_key\_table(key):

key = remove\_spaces(to\_lower\_case(key))

key = key.replace('j', 'i')

key = ''.join(dict.fromkeys(key)) # Remove duplicate letters

alphabet = "abcdefghiklmnopqrstuvwxyz" # 'j' is excluded

key\_table = [c for c in key if c in alphabet]

for char in alphabet:

if char not in key\_table:

key\_table.append(char)

key\_table = np.array(key\_table).reshape(5, 5)

return key\_table

def find\_positions(char1, char2, key\_table):

row1, col1 = np.where(key\_table == char1)

row2, col2 = np.where(key\_table == char2)

return row1[0], col1[0], row2[0], col2[0]

def encrypt\_playfair(plaintext, key\_table):

plaintext = remove\_spaces(to\_lower\_case(plaintext)).replace('j', 'i')

# Make plaintext length even by padding with 'x' if needed

if len(plaintext) % 2 != 0:

plaintext += 'x'

# Split plaintext into pairs of characters

pairs = []

i = 0

while i < len(plaintext):

char1 = plaintext[i]

char2 = plaintext[i + 1] if i + 1 < len(plaintext) else 'x'

if char1 == char2:

pairs.append((char1, 'x'))

i += 1

else:

pairs.append((char1, char2))

i += 2

# Encrypt each pair

encrypted\_text = ""

for char1, char2 in pairs:

row1, col1, row2, col2 = find\_positions(char1, char2, key\_table)

if row1 == row2: # Same row

encrypted\_text += key\_table[row1][(col1 + 1) % 5]

encrypted\_text += key\_table[row2][(col2 + 1) % 5]

elif col1 == col2: # Same column

encrypted\_text += key\_table[(row1 + 1) % 5][col1]

encrypted\_text += key\_table[(row2 + 1) % 5][col2]

else: # Rectangle

encrypted\_text += key\_table[row1][col2]

encrypted\_text += key\_table[row2][col1]

return encrypted\_text

# Server setup

HOST = '127.0.0.1'

PORT = 65432

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as server\_socket:

server\_socket.bind((HOST, PORT))

server\_socket.listen()

print(f"Server listening on {HOST}:{PORT}...")

while True:

conn, addr = server\_socket.accept()

with conn:

print(f"Connected by {addr}")

data = conn.recv(1024).decode()

if data:

plaintext, key = data.split('|')

key\_table = generate\_key\_table(key)

encrypted\_text = encrypt\_playfair(plaintext, key\_table)

conn.sendall(encrypted\_text.encode())

**CLIENT SIDE CODE**

import socket

# Client setup

HOST = '127.0.0.1'

PORT = 65432

plaintext = input("Enter the plaintext: ")

key = input("Enter the key: ")

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as client\_socket:

client\_socket.connect((HOST, PORT))

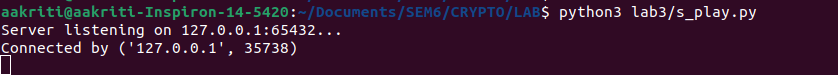
client\_socket.sendall(f"{plaintext}|{key}".encode())

encrypted\_text = client\_socket.recv(1024).decode()

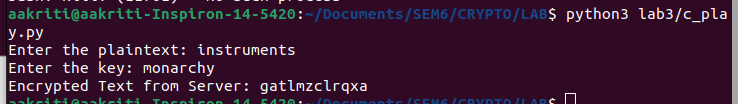
print(f"Encrypted Text from Server: {encrypted\_text}")

**OUTPUT-**

SERVER SIDE



CLIENT SIDE



**TRANSPOSITION COMPLEX**

**SERVER CODE**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

// Function to perform Columnar Transposition Encryption

void columnarEncrypt(char \*plaintext, char \*ciphertext, int \*key, int key\_len)

{

int text\_len = strlen(plaintext);

int rows = (text\_len + key\_len - 1) / key\_len; // Calculate the number of rows

char matrix[rows][key\_len];

int index = 0;

// Fill the matrix row by row

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < key\_len; j++)

{

if (index < text\_len)

{

matrix[i][j] = plaintext[index++];

}

else

{

matrix[i][j] = 'X'; // Fill empty spaces with 'X'

}

}

}

// Read the matrix column by column according to the key

index = 0;

for (int k = 0; k < key\_len; k++)

{

int col = key[k] - 1; // Convert key to zero-based column index

for (int i = 0; i < rows; i++)

{

ciphertext[index++] = matrix[i][col];

}

}

ciphertext[index] = '\0'; // Null-terminate the ciphertext

}

int main()

{

int server\_fd, new\_socket;

struct sockaddr\_in address;

int addrlen = sizeof(address);

char buffer[BUFFER\_SIZE] = {0};

char ciphertext[BUFFER\_SIZE] = {0};

int key[BUFFER\_SIZE];

int key\_len;

// Create socket

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0)

{

perror("Socket failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Bind the socket

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0)

{

perror("Bind failed");

exit(EXIT\_FAILURE);

}

// Listen for connections

if (listen(server\_fd, 3) < 0)

{

perror("Listen failed");

exit(EXIT\_FAILURE);

}

printf("Server is listening on port %d...\n", PORT);

// Accept client connection

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t \*)&addrlen)) < 0)

{

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// Receive key length

read(new\_socket, &key\_len, sizeof(key\_len));

// Receive key from client

read(new\_socket, key, key\_len \* sizeof(int));

// Receive plaintext from client

read(new\_socket, buffer, BUFFER\_SIZE);

printf("Plaintext received: %s\n", buffer);

// Encrypt plaintext

columnarEncrypt(buffer, ciphertext, key, key\_len);

// Send ciphertext back to client

send(new\_socket, ciphertext, strlen(ciphertext), 0);

printf("Ciphertext sent: %s\n", ciphertext);

close(new\_socket);

close(server\_fd);

return 0;

}

**CLIENT CODE**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main()

{

int sock = 0;

struct sockaddr\_in serv\_addr;

char plaintext[BUFFER\_SIZE] = {0};

char buffer[BUFFER\_SIZE] = {0};

int key[BUFFER\_SIZE];

int key\_len;

// Create socket

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

printf("\nSocket creation error\n");

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary form

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0)

{

printf("\nInvalid address/Address not supported\n");

return -1;

}

// Connect to server

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)

{

printf("\nConnection Failed\n");

return -1;

}

// Input plaintext

printf("Enter the plaintext: ");

fgets(plaintext, BUFFER\_SIZE, stdin);

plaintext[strcspn(plaintext, "\n")] = '\0'; // Remove newline character

// Input key

printf("Enter the key length: ");

scanf("%d", &key\_len);

printf("Enter the key (e.g., for '4312567', enter '4 3 1 2 5 6 7'): ");

for (int i = 0; i < key\_len; i++)

{

scanf("%d", &key[i]);

}

// Send key length to server

send(sock, &key\_len, sizeof(key\_len), 0);

// Send key to server

send(sock, key, key\_len \* sizeof(int), 0);

// Send plaintext to server

send(sock, plaintext, strlen(plaintext), 0);

// Receive ciphertext from server

read(sock, buffer, BUFFER\_SIZE);

printf("Ciphertext received: %s\n", buffer);

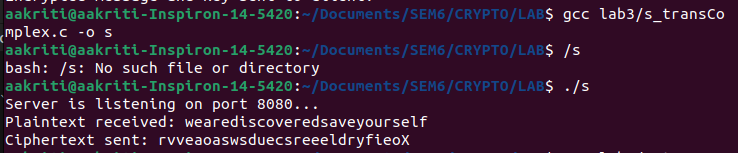
close(sock);

return 0;

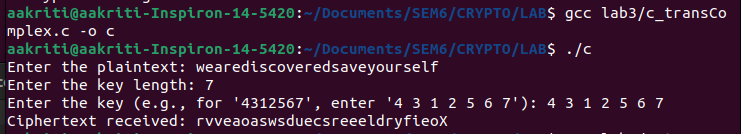
}

**OUTPUT:-**

**SERVER**

****

**CLIENT**

****