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:

Mann Patel

22BCP107

G3 batch

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. 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. b) Encrypt the input words PLAINTEXT= RAG BABY to obtain CIPHERTEXT = SCJ DDFD		

Experiment 1 **Date: 23rd July 2024**

AIM: Study and implement a program for Caesar Cipher

Original Approach

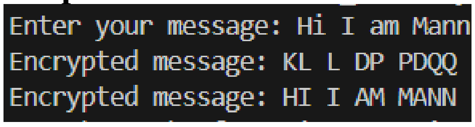
Introduction:
The Caesar cipher is a simple encryption technique that was used by Julius Caesar to send secret messages to his allies. It works by shifting the letters in the plaintext message by a certain number of positions, known as the “shift” or “key”. The Caesar Cipher technique is one of the earliest and simplest methods of encryption techniques.

Example:
It’s simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with a fixed number of positions down the alphabet. For example, with a shift of 1, A would be replaced by B, B would become C, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.

Source Code:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

//function to find index of character
int find_ch(char ch, char alphabets[]){
    for (int i=0 ; i< 26; i++){
        if (ch == alphabets[i]){
            return i;
        }
    }
    return -1;
}

//function to perform encryption on message
void encrypt(char message[], char alphabets[], char result[]){

    int key = 3;
    char ch;
    int index;
    int len = strlen(message);

    for(int i = 0; i<len; i++){
        ch = message[i];
        index = find_ch(ch,alphabets);
        if (index == -1){
            result[i] = ' ';
        }
        else{
            index = (index + key)%26;
            result[i] = alphabets[index];
        }
    }
    result[len] = '\0';
}

//function to perform decryption on message
void decrypt(char message[], char alphabets[], char result[]){

    int key = 3;
    char ch;
    int index;
    int len = strlen(message);

    for(int i = 0; i<len; i++){
        ch = message[i];
        index = find_ch(ch,alphabets);
        if (index == -1){
            result[i] = ' ';
        }
        else{
            index = (index - key + 26)%26;
            result[i] = alphabets[index];
        }
    }
    result[len] = '\0';
}

void main () {

    //declaring all alphabets
    char alphabets[26] = {'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'};

    //declaring string for storing data
    char message[100];
    char encrypt_result[100];
    char decrypt_result[100];

    //getting message from user
    printf("Enter your message: ");
    gets(message);

    strupr(message);

    //performing encryption on original message
    encrypt(message, alphabets, encrypt_result);
    printf("Encrypted message: ");
    puts(encrypt_result);

    //performing decryption on encryption message
    decrypt(encrypt_result, alphabets, decrypt_result);
    printf("Encrypted message: ");
    puts(decrypt_result);

}
```

Output screenshot:



```
Enter your message: Hi I am Mann
Encrypted message: KL L DP PDQQ
Encrypted message: HI I AM MANN
```

Revised Approach

Introduction:

For a revised approach, I considered complicating the program in such a way that even if our original text contains the same letter multiple times, in the encrypted text, those letters are converted into more than one or two different letters. So, this approach shouldn’t be based on the alphabet itself but instead should focus on the position of the alphabet in the original string. I divided the message into three parts and applied encryption to each part sequentially. Each time, the encryption key is a multiple of 2 to increase randomness, avoiding repetition with a constant key.

Example:

Original Message: Mann
Encryption Process:

1st Loop – OANN {key = 2}
2nd Loop – SENN {key = 4}
3rd Loop – AMVV {key = 8}

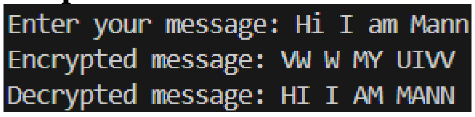
Encrypted Message: AMVV

Source Code:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

//function to find index of character
int find_ch(char ch, char alphabets[]){
    for (int i=0 ; i< 26; i++){
        if (ch == alphabets[i]){
            return i;
        }
    }
    return -1;
}

//function to perform encryption on message
void encrypt(char message[], char alphabets[], char result[]){
    int parts = 3;
    int length = strlen(message);
    int index;
    int partition;
    int key = 2;
    char ch;
    if (length < 3){
        for(int i=0; i<length ; i++){
            ch = message[i];
            index = find_ch(ch,alphabets);
            if (index == -1){
                result[i] = ' ';
            }
            else{
                index = (index + key)%26;
                result[i] = alphabets[index];
            }
        }
    }
    else{
        partition = length/parts;
        int round = 1;
        while (parts!=0){
            int i;
            for (i = 0; i<round*partition; i++){
                ch = message[i];
                index = find_ch(ch,alphabets);
                if (index == -1){
                    result[i] = ' ';
                }
                else{
                    index = (index + key)%26;
                    result[i] = alphabets[index];
                    message[i] = result[i];
                }
            }

            if ( i<length && parts == 1){
                for(i = i; i<length; i++){
                    ch = message[i];
                    index = find_ch(ch,alphabets);
                    if (index == -1){
                        result[i] = ' ';
                    }
                    else{
                        index = (index + key)%26;
                        result[i] = alphabets[index];
                        message[i] = result[i];
                    }
                }
            }

            round++;
            key = key*2;
```

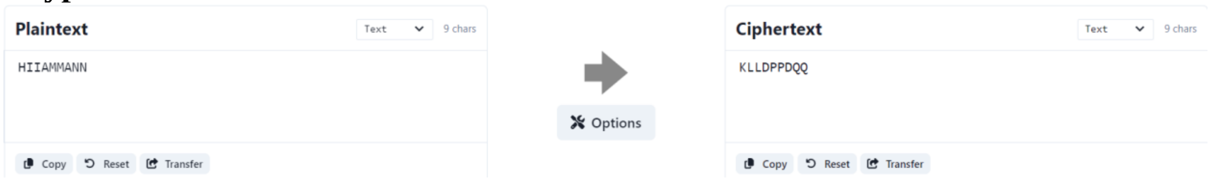
```c
            parts--;
        }
    }
    result[length] = '\0';
}

//function to perform decryption on message
void decrypt(char message[], char alphabets[], char result[]){
    int parts = 3;
    int length = strlen(message);
    int index;
    int partition;
    int key = 2;
    char ch;

    if (length < 3){
        for(int i=0; i<length ; i++){
            ch = message[i];
            index = find_ch(ch,alphabets);
            if (index == -1){
                result[i] = ' ';
            }
            else{
                index = (index - key + 26)%26;
                result[i] = alphabets[index];
            }
        }
    }
    else{
        partition = length/parts;
        int round = 3;
        key = 8;
        while (parts!=0){
            int i = 0;
            for (i = 0; i<round*partition && parts != 3; i++){
                ch = message[i];
                index = find_ch(ch,alphabets);
                if (index == -1){
                    result[i] = ' ';
                }
                else{
                    index = (index - key + 26)%26;
                    result[i] = alphabets[index];
                    message[i] = result[i];
                }
            }

            if ( i<length && parts == 3){
                for(i = 0; i<length; i++){
                    ch = message[i];
                    index = find_ch(ch,alphabets);
                    if (index == -1){
                        result[i] = ' ';
                    }
                    else{
                        index = ((index - key + 26)%26);
                        result[i] = alphabets[index];
                        message[i] = result[i];
                    }
                }
            }

            round--;
            key = key/2;
            parts--;
        }
    }
    result[length] = '\0';
}

void main () {

    //declaring all alphabets
    char alphabets[26] = {'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'};

    //declaring string for storing data
    char message[100];
    char encrypt_result[100];
    char decrypt_result[100];

    //getting message from user
    printf("Enter your message: ");
    gets(message);

    strupr(message);

    //performing encryption on original message
    encrypt(message, alphabets, encrypt_result);
    printf("Encrypted message: ");
    puts(encrypt_result);

    //performing decryption on encryption message
    decrypt(encrypt_result, alphabets, decrypt_result);
    printf("Decrypted message: ");
    puts(decrypt_result);

}
```

Output screenshot:



```
Enter your message: Hi I am Mann
Encrypted message: VW W MY UIVV
Decrypted message: HI I AM MANN
```

Cryptool screenshot:



```
HIIAMMANN
```

```
KLLDPPDQQ
```

Comparative Analysis of original and revised approaches:

For original approach:
function find_ch runs for 26 times and encrypt function runs until string ends.
Time Complexity: O(26*n) = O(n), where n is length of string.

For revised approach:
function find_ch runs for 26 times and for encryption outer loop will run for 3 times and inner loop runs maximum until string ends and if length is less than 3 it will run same as original approach.
Time Complexity: O(26*3*n) = O(n), where n is length of string.

Based on performance, the revised approach is much better than the original approach due to increased randomness and the generation of repeated words during encryption.

Conclusion:

The revised approach demonstrates a significant improvement in encryption complexity by introducing increased randomness and generating different letters for repeated characters, making it more secure than the original Caesar cipher technique.

References:

Rohit Singh, Naveen Kumar, Anuja A review paper on cryptography of modified Caesar Cipher – August 2018.
Manepalli Dharani Pujitha Security Enhancement Using Caesar Cipher – November 2022

Experiment 2 Date: 30th July 2024

AIM: Study and implement a program for 5x5 Playfair Cipher to encrypt and decrypt the message.

Original Approach

Introduction:

The Playfair Cipher, devised by Charles Wheatstone in 1854 and popularized by Lord Playfair, encrypts messages by pairing letters and using a 5x5 matrix filled with a keyword followed by the remaining alphabet (combining 'I' and 'J' to fit). To handle duplicate letters or single leftover letters, a filler character is introduced.

Encryption rules:

- Same Row: Shift each letter one position to the right.

- Same Column: Shift each letter one position downward.

- Different Row and Column: Replace each letter with the one in its own row and the other letter's column.

Example:

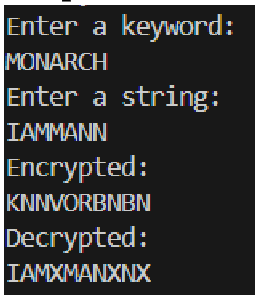
Let's say we want to convert the word "mann" to a cipher text using the playfair cipher, and the keyword which we'll be using is "nice". So, first we make a 5 X 5 matrix and place the alphabets "n", "i", "c", "e" respectively. After that we'll add the remaining of the alphabet starting through A to Z and making sure no duplicity occurs in the matrix.
After this, we make the pairs of the message/ word and if there's any duplicity in the pair or there's no other letter left to pair the remaining one out, we use X as our filler. So, we make the pairs as – MA, NX, NX. Now we match the alphabets according to the rules above and we get "PCCVCV" as out decrypted word.

Source Code:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void removeSpace(char* message){
    char string[100];
    int k=0;

    for (int i=0; i<strlen(message); i++){
        if (message[i] != ' ' && message[i] != 'J') {
            string[k] = message[i];
            k++;
        }
        else if (message[i] == 'J') {
            string[k] = 'I';
            k++;
        }
    }
    string[k] = '\0';
    strcpy(message,string);
}

void prepareKeyword(char* keyword) {
    int len = strlen(keyword);
    int index = 0;

    for (int i = 0; i < len; i++) {
        if (keyword[i] != 'J') {
            keyword[index++] = keyword[i];
        }
        else{
            keyword[index++] = 'I';
        }
    }

    keyword[index] = '\0';
}

void matrixtable(char* keyword, char matrix[5][5]){
    int seen[26] = {0};
    int row = 0, col = 0;

    prepareKeyword(keyword);

    for (int i = 0; keyword[i] != '\0'; i++) {
        if (!seen[keyword[i] - 'A']) {
            matrix[row][col++] = keyword[i];
            seen[keyword[i] - 'A'] = 1;
            if (col == 5) {
                col = 0;
                row++;
            }
        }
    }

    for (char ch = 'A'; ch <= 'Z'; ch++) {
        if (ch != 'J' && !seen[ch - 'A']) {
            matrix[row][col++] = ch;
            seen[ch - 'A'] = 1;
            if (col == 5) {
                col = 0;
                row++;
            }
        }
    }
}

void findcharacter(char ch, int* row, int* col, char matrix[5][5]){
    for (int i=0; i<5; i++){
        for (int j=0; j<5; j++){
            if (matrix[i][j] == ch){
                *row = i;
                *col = j;
                break;
            }
        }
    }
}

void encryptpair(char *ch1, char *ch2, char matrix[5][5]){
    int row1, row2, col1, col2;

    findcharacter(*ch1, &row1, &col1, matrix);
    findcharacter(*ch2, &row2, &col2, matrix);

    if(row1 == row2){
        *ch1 = matrix[row1][(col1 + 1) % 5];
        *ch2 = matrix[row2][(col2 + 1) % 5];
    }
    else if(col1 == col2){
        *ch1 = matrix[(row1 + 1) % 5][col1];
        *ch2 = matrix[(row2 + 1) % 5][col2];
    }
    else{
        *ch1 = matrix[row1][col2];
        *ch2 = matrix[row2][col1];
    }
}

void playcipherencrypt(char* message, char matrix[5][5]){
    int len = strlen(message);
    char modified_message[3 * len ];
    char ch1, ch2;
    int i=0;
    int k=0;
    for (i = 0; i < len; i+=2) {
        ch1 = message[i];
        if (i + 1 >= len) {
            ch2 = 'X';
        } else {
            ch2 = message[i + 1];
        }
```

```c
        if (ch1 == ch2) {
            ch2 = 'X';
            i--;
        }

        encryptpair(&ch1, &ch2, matrix);
        modified_message[k] = ch1;
        modified_message[k + 1] = ch2;
        k+=2;
    }

    modified_message[k] = '\0';
    strcpy(message,modified_message);
}

void decryptpair(char *ch1, char *ch2, char matrix[5][5]) {
    int row1, row2, col1, col2;

    findcharacter(*ch1, &row1, &col1, matrix);
    findcharacter(*ch2, &row2, &col2, matrix);

    if(row1 == row2) {
        *ch1 = matrix[row1][(col1 - 1 + 5) % 5];
        *ch2 = matrix[row2][(col2 - 1 + 5) % 5];
    }
    else if(col1 == col2) {
        *ch1 = matrix[(row1 - 1 + 5) % 5][col1];
        *ch2 = matrix[(row2 - 1 + 5) % 5][col2];
    }
    else {
        *ch1 = matrix[row1][col2];
        *ch2 = matrix[row2][col1];
    }
}

void playcipherdecrypt(char* message, char matrix[5][5]) {
    int len = strlen(message);
    char modified_message[len + 1];
    char ch1, ch2;
    int i = 0;
    int k = 0;

    for(i=0;i<len;i+=2) {
        ch1 = message[i];
        if (i + 1 >= len) {
            ch2 = 'X';
        } else {
            ch2 = message[i + 1];
        }

        decryptpair(&ch1, &ch2, matrix);
        modified_message[k] = ch1;
        modified_message[k + 1] = ch2;

        k += 2;
    }

    modified_message[k] = '\0';
    strcpy(message, modified_message);
}

void main(){
    char keyword[50];
    char message[100];
    char matrix[5][5];

    puts("Enter a keyword: ");
    gets(keyword);
    strupr(keyword);

    matrixtable(keyword,matrix);

    puts("Enter a string: ");
    gets(message);
    strupr(message);
    removeSpace(message);

    playcipherencrypt(message,matrix);

    puts(message);

    playcipherdecrypt(message,matrix);

    puts(message);

}
```

Output screenshot:

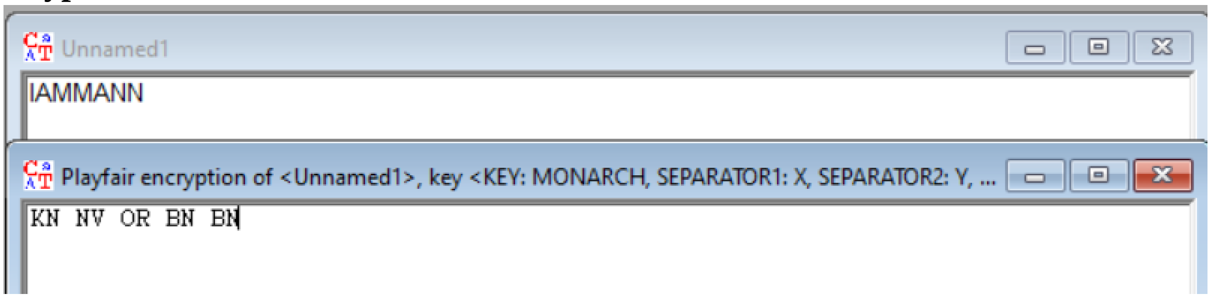


```
Enter a keyword:
MONARCH
Enter a string:
IAMMANN
Encrypted:
KNNVORBNBN
Decrypted:
IAMXMANXNX
```

Cryptanalysis:

The Playfair cipher is particularly vulnerable to known plaintext attacks due to its deterministic encryption process and fixed 5x5 key table, hence allowing the analyst to know both the plaintext and the ciphertext. This structure allows attackers to easily deduce the key from plaintext-ciphertext pairs. The key tables have 25! which is not possible to break using brute force attacks because of the vast number of combinations renders them practically impractical. While the cipher can also be susceptible to ciphertext-only and chosen plaintext attacks, these methods require more complex analysis and additional resources. Ciphertext-only attacks involve extensive statistical analysis, while chosen plaintext attacks depend on the attacker's ability to select specific plaintexts and obtain their corresponding ciphertexts. Consequently, known plaintext attacks are the most direct and effective approach to breaking the Playfair cipher, given its predictable encryption scheme and manageable key space.

Cryptool screenshot:



Unnamed1

```
IAMMANN
```

Playfair encryption of <Unnamed1>, key <KEY: MONARCH, SEPARATOR1: X, SEPARATOR2: Y, ...

```
KN NV OR BN BN
```

<u>Revised Approach</u>

Introduction:

The revised implementation of the Playfair cipher introduces an advanced modification to the traditional encryption process. By dividing both the keyword and the message into two distinct parts, a dual encryption approach is employed. The first part of the message is encrypted using the second part of the keyword matrix, while the second part of the message is encrypted with the first part of the keyword matrix. This technique enhances the complexity and security of the cipher, making it more resilient to cryptographic analysis. The decryption process mirrors this approach, ensuring accurate retrieval of the original message. This method demonstrates an innovative enhancement to the Playfair cipher, providing a more robust encryption mechanism.

Example:

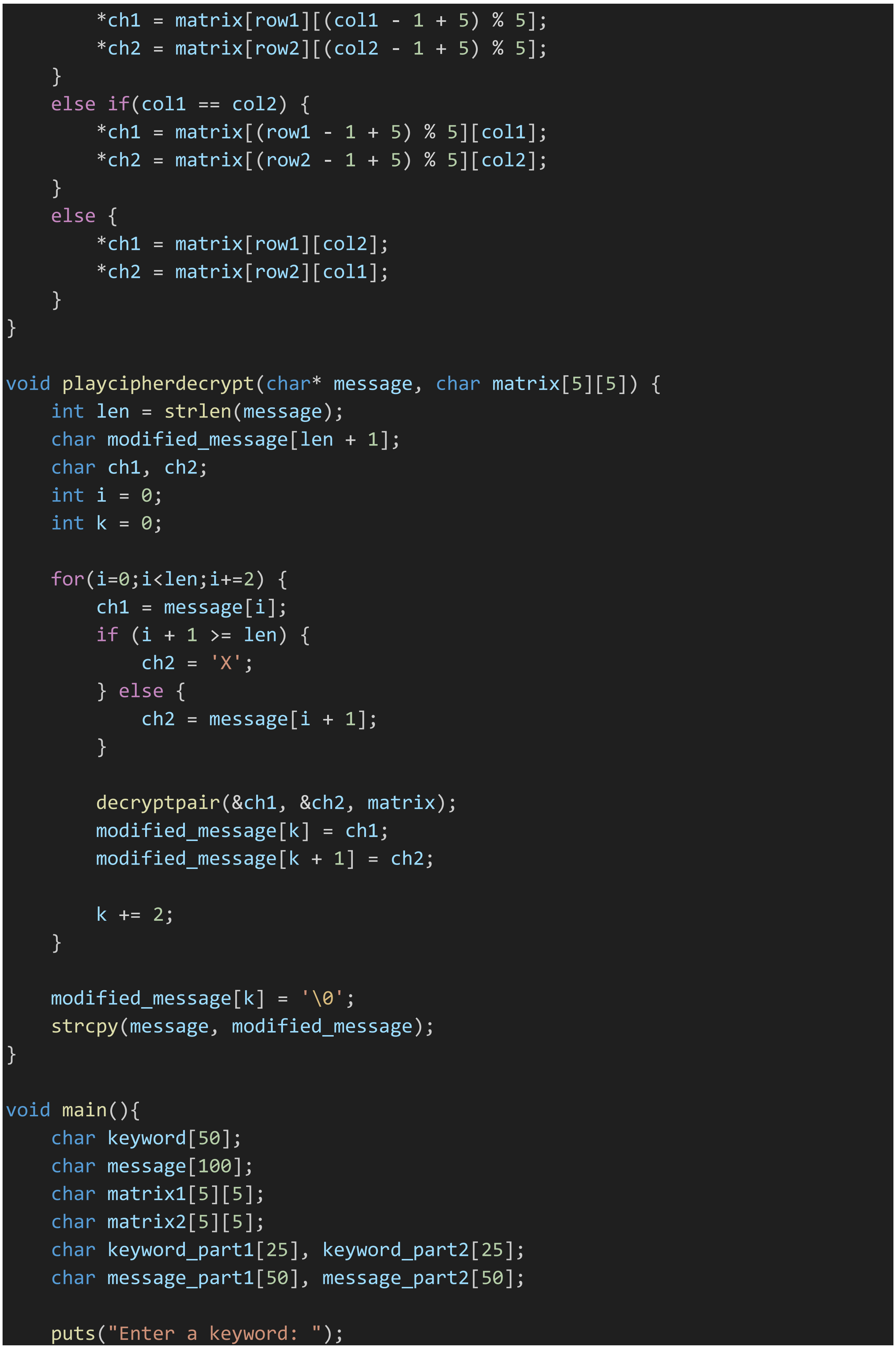
To encrypt the word "mann" using the modified Playfair cipher with the keyword "nice," the

keyword is split into “ni” and “ce.” Two 5x5 matrices are created: one using “ni” and the other using “ce.” The word “mann” is split into pairs: “ma” and “nn.” If any pair has duplicate letters or a single letter remains, “X” is used as a filler.
Encrypting the pairs:
-MA converts to NE
-NN converts to AVAV

Source Code:

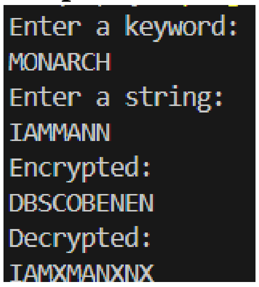
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void removeSpace(char* message){
    char string[100];
    int k=0;

    for (int i=0; i<strlen(message); i++){
        if (message[i] != ' ' && message[i] != 'J') {
            string[k] = message[i];
            k++;
        }
        else if (message[i] == 'J') {
            string[k] = 'I';
            k++;
        }
    }
    string[k] = '\0';
    strcpy(message,string);
}

void prepareKeyword(char* keyword) {
    int len = strlen(keyword);
    int index = 0;

    for (int i = 0; i < len; i++) {
        if (keyword[i] != 'J') {
            keyword[index++] = keyword[i];
        }
        else{
            keyword[index++] = 'I';
        }
    }

    keyword[index] = '\0';
}

void matrixtable(char* keyword, char matrix[5][5]){
```

```c
    int seen[26] = {0};
    int row = 0, col = 0;

    prepareKeyword(keyword);

    for (int i = 0; keyword[i] != '\0'; i++) {
        if (!seen[keyword[i] - 'A']) {
            matrix[row][col++] = keyword[i];
            seen[keyword[i] - 'A'] = 1;
            if (col == 5) {
                col = 0;
                row++;
            }
        }
    }

    for (char ch = 'A'; ch <= 'Z'; ch++) {
        if (ch != 'J' && !seen[ch - 'A']) {
            matrix[row][col++] = ch;
            seen[ch - 'A'] = 1;
            if (col == 5) {
                col = 0;
                row++;
            }
        }
    }
}

void findcharacter(char ch, int* row, int* col, char matrix[5][5]){
    for (int i=0; i<5; i++){
        for (int j=0; j<5; j++){
            if (matrix[i][j] == ch){
                *row = i;
                *col = j;
                break;
            }
        }
    }
}

void encryptpair(char *ch1, char *ch2, char matrix[5][5]){
    int row1, row2, col1, col2;

    findcharacter(*ch1, &row1, &col1, matrix);
    findcharacter(*ch2, &row2, &col2, matrix);

    if(row1 == row2){
        *ch1 = matrix[row1][(col1 + 1) % 5];
        *ch2 = matrix[row2][(col2 + 1) % 5];
    }
    else if(col1 == col2){
        *ch1 = matrix[(row1 + 1) % 5][col1];
        *ch2 = matrix[(row2 + 1) % 5][col2];
    }
    else{
        *ch1 = matrix[row1][col2];
        *ch2 = matrix[row2][col1];
    }
}

void playcipherencrypt(char* message, char matrix[5][5]){
    int len = strlen(message);
    char modified_message[3 * len ];
    char ch1, ch2;
    int i=0;
    int k=0;
    for (i = 0; i < len; i+=2) {
        ch1 = message[i];
        if (i + 1 >= len) {
            ch2 = 'X';
        } else {
            ch2 = message[i + 1];
        }

        if (ch1 == ch2) {
            ch2 = 'X';
            i--;
        }

        encryptpair(&ch1, &ch2, matrix);
        modified_message[k] = ch1;
        modified_message[k + 1] = ch2;
        k+=2;
    }

    modified_message[k] = '\0';
    strcpy(message,modified_message);
}

void decryptpair(char *ch1, char *ch2, char matrix[5][5]) {
    int row1, row2, col1, col2;

    findcharacter(*ch1, &row1, &col1, matrix);
    findcharacter(*ch2, &row2, &col2, matrix);

    if(row1 == row2) {
```



```c
        *ch1 = matrix[row1][(col1 - 1 + 5) % 5];
        *ch2 = matrix[row2][(col2 - 1 + 5) % 5];
    }
    else if(col1 == col2) {
        *ch1 = matrix[(row1 - 1 + 5) % 5][col1];
        *ch2 = matrix[(row2 - 1 + 5) % 5][col2];
    }
    else {
        *ch1 = matrix[row1][col2];
        *ch2 = matrix[row2][col1];
    }
}

void playcipherdecrypt(char* message, char matrix[5][5]) {
    int len = strlen(message);
    char modified_message[len + 1];
    char ch1, ch2;
    int i = 0;
    int k = 0;

    for(i=0;i<len;i+=2) {
        ch1 = message[i];
        if (i + 1 >= len) {
            ch2 = 'X';
        } else {
            ch2 = message[i + 1];
        }

        decryptpair(&ch1, &ch2, matrix);
        modified_message[k] = ch1;
        modified_message[k + 1] = ch2;

        k += 2;
    }

    modified_message[k] = '\0';
    strcpy(message, modified_message);
}

void main(){
    char keyword[50];
    char message[100];
    char matrix1[5][5];
    char matrix2[5][5];
    char keyword_part1[25], keyword_part2[25];
    char message_part1[50], message_part2[50];

    puts("Enter a keyword: ");
    gets(keyword);
    strupr(keyword);

    int len_keyword = strlen(keyword);
    int mid_keyword = len_keyword / 2;
    strncpy(keyword_part1, keyword, mid_keyword);
    keyword_part1[mid_keyword] = '\0';
    strcpy(keyword_part2, keyword + mid_keyword);

    matrixtable(keyword_part1,matrix1);
    matrixtable(keyword_part2,matrix2);

    puts("Enter a string: ");
    gets(message);
    strupr(message);
    removeSpace(message);

    int len_message = strlen(message);
    int mid_message = len_message / 2;
    strncpy(message_part1, message, mid_message);
    message_part1[mid_message] = '\0';
    strcpy(message_part2, message + mid_message);

    playcipherencrypt(message_part2, matrix1);
    playcipherencrypt(message_part1, matrix2);

    strcpy(message,message_part1);
    strcat(message,message_part2);
    puts("Encrypted: ");
    puts(message);

    playcipherdecrypt(message_part1, matrix2);
    playcipherdecrypt(message_part2, matrix1);

    strcpy(message, message_part1);
    strcat(message, message_part2);
    puts("Decrypted: ");
    puts(message);

}
```

Output screenshot:



```
Enter a keyword:
MONARCH
Enter a string:
IAMMANN
Encrypted:
DBSCOBENEN
Decrypted:
IAMXMANXNX
```

Comparative Analysis of original and revised approaches:

The original approach uses a single 5x5 matrix for encryption and decryption, which makes it vulnerable to frequency analysis attacks. Since each character in the message corresponds to a unique character in the matrix, patterns in the plaintext can be more easily detected in the ciphertext. In contrast, the revised approach splits both the keyword and the message into two parts and uses two different matrices for encryption and decryption. This method introduces more complexity and variability in the encrypted message, making it more resistant to frequency analysis and other cryptographic attacks. By using two matrices, the revised approach ensures that identical pairs of characters in different parts of the message do not result in identical ciphertext, thereby enhancing security.

Conclusion:

In conclusion, the revised approach significantly improves the security of the cipher by using two matrices, which disrupts patterns and reduces the risk of frequency analysis attacks. This added complexity makes it more difficult for attackers to decipher the message. Overall, the revised method offers a more robust encryption scheme compared to the original.

References:

R.Deepthi A Survey Paper on Playfair Cipher and its Variants – April 2017
Nikhil Sharma, Himmat Meghwal, Munish Mehta, Tarun Kumar A Review on Playfair Substitution Cipher and Frequency Analysis Attack on Playfair – May 2018

<u>Experiment 3</u> **Date: 20th August 2024**

AIM: Study and implement a program for Rail Fence Cipher with columnar transposition

<u>Original Approach</u>

Introduction:

The rail fence cipher (also called a zigzag cipher) is a classical type of transposition cipher. It derives its name from the manner in which encryption is performed, in analogy to a fence built with horizontal rails.

Encryption: In the rail fence cipher, the plaintext is written downwards diagonally on successive "rails" of an imaginary fence, then moving up when the bottom rail is reached, down again when the top rail is reached, and so on until the whole plaintext is written out. The ciphertext is then read off in rows.

Decryption: To decrypt a rail fence cipher, start by creating an empty matrix with the same number of rails used in the encryption. Fill this matrix with the ciphertext characters in a zigzag pattern, moving down and up across the rails. Once the matrix is filled, read the characters in the same zigzag order to reconstruct the original plaintext.

Example:

To encrypt a word “HELLO”, this type of matrix is made using key 3. Now we will read it in zigzag manner.
Encrypt message: HOELL
Matrix:
H _ _ _ O

_ E _ L _

_ _ L _ _

Now to decrypt the encrypted message “HOELL”, we will first construct the similar matrix in zigzag manner and we read it in zigzag manner to get the plaintext.

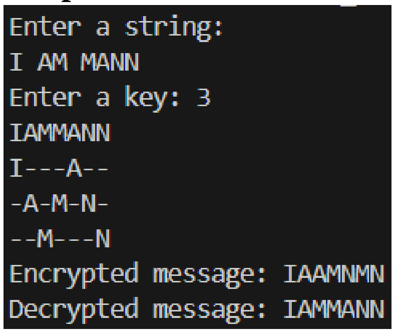
Decrypt message: HELLO

Source Code:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

void removeSpace(char* message){
    char string[100];
    int k=0;

    for (int i=0; i<strlen(message); i++){
        if (message[i] != ' ') {
            string[k] = message[i];
            k++;
        }
    }
    string[k] = '\0';
    strcpy(message,string);
}

void makematrix_encrypt(char* string, int key, char
matrix[key][strlen(string)]) {
    bool isreverse = false;
    int row = 0;
    int col = 0;

    while (col < strlen(string)){
        matrix[row][col] = string[col];
        if (!isreverse){
            row++;
            if(row == key){
                isreverse=true;
                row-=2;
            }
        }
        else {
            row--;
            if(row == -1){
                isreverse = false;
                row+=2;
            }
        }
        col++;
    }
}

void makematrix_decrypt(char* string, int key, char
matrix[key][strlen(string)]) {
    bool isreverse = false;
    int row = 0;
    int col = 0;

    while (col < strlen(string)){
        matrix[row][col] = '*';
        if (!isreverse){
            row++;
            if(row == key){
                isreverse=true;
                row-=2;
            }
        }
        else {
```

```c
            row--;
            if(row == -1){
                isreverse = false;
                row+=2;
            }
        }
        col++;
    }
}

void railfence_encrypt(char* message, char* encrypt_message, int key, char
matrix[key][strlen(message)]){
    makematrix_encrypt(message, key, matrix);
    for (int i=0; i<key; i++){
        for (int j=0; j<strlen(message); j++){
            printf("%c",matrix[i][j]);
        }
        printf("\n");
    }

    int k=0;
    int increase = (key-1)*2;

    for (int i=0 ; i<key; i++){
        for (int j=0; j<strlen(message); j++){
            if (matrix[i][j] != '-'){
                encrypt_message[k++] = matrix[i][j];
            }
        }
    }

    encrypt_message[k] = '\0';
    printf("Encrypted message: %s\n",encrypt_message);
}

void railfence_decrypt(char* message, char* decrypt_message, int key, char
matrix[key][strlen(message)]){

    makematrix_decrypt(message, key, matrix);
    int k = 0;

    for (int i=0 ; i<key; i++){
        for (int j=0; j<strlen(message); j++){
            if (matrix[i][j] == '*'){
                matrix[i][j] = message[k++];
            }
        }
    }
    bool isreverse = false;
    int row = 0;
    int col = 0;
    k = 0;

    while (col < strlen(message)){
        decrypt_message[k++] = matrix[row][col];
        if (!isreverse){
            row++;
            if(row == key){
                isreverse=true;
                row-=2;
            }
        }
        else {
            row--;
            if(row == -1){
                isreverse = false;
                row+=2;
            }
        }
        col++;
    }
    decrypt_message[k] = '\0';
    printf("Decrypted message: %s",decrypt_message);
}

void main(){
    int key;
    char message[100];
    char encrypt_message[100];
    char decrypt_message[100];

    puts("Enter a string: ");
    gets(message);
    strupr(message);
    removeSpace(message);

    printf("Enter a key: ");
    scanf("%d", &key);

    puts(message);

    char matrix_encrypt[key][strlen(message)];
    char matrix_decrypt[key][strlen(message)];

    for (int i = 0; i < key; i++) {
        for (int j = 0; j < strlen(message); j++) {
            matrix_encrypt[i][j] = '-';
        }
    }

    railfence_encrypt(message, encrypt_message, key, matrix_encrypt);
    railfence_decrypt(encrypt_message, decrypt_message, key, matrix_decrypt);

}
```

Output screenshot:

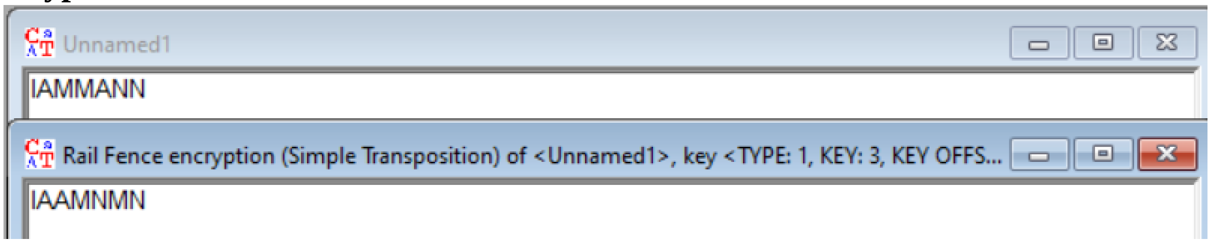


```
Enter a string:
I AM MANN
Enter a key: 3
IAMMANN
I---A--
-A-M-N-
--M---N
Encrypted message: IAAMNMN
Decrypted message: IAMMANN
```

Cryptanalysis:

The rail fence cipher is typically broken using brute force, as the number of rows, which serves as the key for the matrix, is limited. By iterating through all possible row counts, the secret message can eventually be revealed. The rail fence cipher can also be broken using a known plaintext attack. If a portion of the original message is known, the key (number of rows) can be determined by analysing how the plaintext aligns with the encrypted message. Once the key is identified, the entire secret message can be decrypted accurately.

Cryptool screenshot:



Unnamed1

IAMMANN

Rail Fence encryption (Simple Transposition) of <Unnamed1>, key <TYPE: 1, KEY: 3, KEY OFFS...

IAAMNMN

<u>Revised Approach</u>

Introduction:

In the revised approach of the rail fence cipher, instead of reading the rows sequentially from the first to the last, I generate a random array with unique numbers ranging from 0 to key-1. This array is used to create the encrypted message. During decryption, I use the same array to place the characters in a zigzag manner in the matrix, which then reconstructs the proper matrix, allowing me to derive the plaintext.

Example:

To encrypt a word "HELLO", this type of matrix is made using key 3. Now we will read it in

zigzag manner on basis of our array.
Array: 2 0 1
Encrypt message: LHOEL
Matrix:
H _ _ _ O

_ E _ L _

_ _ L _ _

Now to decrypt the encrypted message "LHOEL", we will first construct the similar matrix in zigzag manner on basis of our key and we read it in zigzag manner to get the plaintext.

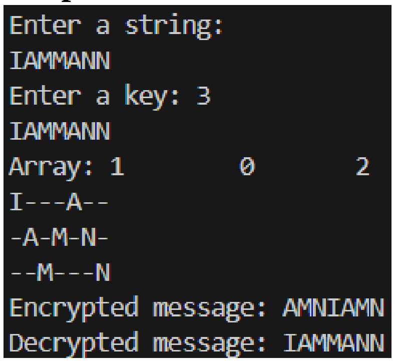
Decrypt message: HELLO

Source Code:

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>

void shuffle(int array[], int n) {
    for (int i = n - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

void removeSpace(char* message){
    char string[100];
    int k=0;

    for (int i=0; i<strlen(message); i++){
        if (message[i] != ' ') {
            string[k] = message[i];
            k++;
        }
    }
    string[k] = '\0';
    strcpy(message,string);
}

void makematrix_encrypt(char* string, int key, char
matrix[key][strlen(string)]) {
    bool isreverse = false;
    int row = 0;
    int col = 0;
    while (col < strlen(string)){
        matrix[row][col] = string[col];
        if (!isreverse){
            row++;
            if(row == key){
                isreverse=true;
                row-=2;
            }
        }
        else {
            row--;
            if(row == -1){
                isreverse = false;
                row+=2;
            }
        }
        col++;
    }
}

void makematrix_decrypt(char* string, int key, char
matrix[key][strlen(string)]) {
    bool isreverse = false;
    int row = 0;
    int col = 0;

    while (col < strlen(string)){
        matrix[row][col] = '*';
        if (!isreverse){
            row++;
            if(row == key){
                isreverse=true;
                row-=2;
            }
        }
        else {
            row--;
            if(row == -1){
                isreverse = false;
                row+=2;
            }
        }
        col++;
    }
}
```

```c
void railfence_encrypt(char* message, char* encrypt_message, int key, char
matrix[key][strlen(message)], int array[key]){

    makematrix_encrypt(message, key, matrix);
    for (int i=0; i<key; i++){
        for (int j=0; j<strlen(message); j++){
            printf("%c",matrix[i][j]);
        }
        printf("\n");
    }

    int k=0;

    for (int i=0 ; i<key; i++){
        for (int j=0; j<strlen(message); j++){
            if (matrix[array[i]][j] != '-'){
                encrypt_message[k++] = matrix[array[i]][j];
            }
        }
    }

    encrypt_message[k] = '\0';
    printf("Encrypted message: %s\n",encrypt_message);
}

void railfence_decrypt(char* message, char* decrypt_message, int key, char
matrix[key][strlen(message)], int array[key]){

    makematrix_decrypt(message, key, matrix);
    int k = 0;

    for (int i=0 ; i<key; i++){
        for (int j=0; j<strlen(message); j++){
            if (matrix[array[i]][j] == '*'){
                matrix[array[i]][j] = message[k++];
            }
        }
    }

    bool isreverse = false;
    int row = 0;
    int col = 0;
    k = 0;

    while (col < strlen(message)){
        decrypt_message[k++] = matrix[row][col];
        if (!isreverse){
            row++;
            if(row == key){
                isreverse=true;
                row-=2;
            }
        }
        else {
            row--;
            if(row == -1){
                isreverse = false;
                row+=2;
            }
        }
        col++;
    }
    decrypt_message[k] = '\0';
    printf("Decrypted message: %s",decrypt_message);
}

void main(){
    int key;
    char message[100];
    char encrypt_message[100];
    char decrypt_message[100];

    puts("Enter a string: ");
    gets(message);
    strupr(message);
    removeSpace(message);

    printf("Enter a key: ");
    scanf("%d", &key);

    puts(message);

    char matrix_encrypt[key][strlen(message)];
    char matrix_decrypt[key][strlen(message)];
    int array[key];

    for (int i = 0; i < key; i++) {
        array[i] = i;
    }

    srand(time(NULL));
    shuffle(array, key);

    printf("Array: ");
    for (int i = 0; i < key; i++) {
        printf("%d\t",array[i]);
    }
    printf("\n");

    for (int i = 0; i < key; i++) {
        for (int j = 0; j < strlen(message); j++) {
            matrix_encrypt[i][j] = '-';
        }
    }

    railfence_encrypt(message, encrypt_message, key, matrix_encrypt, array);
    railfence_decrypt(encrypt_message, decrypt_message, key, matrix_decrypt,
array
    );

}
```

Output screenshot:



```
Enter a string:
IAMMANN
Enter a key: 3
IAMMANN
Array: 1        0       2
I---A--
-A-M-N-
--M---N
Encrypted message: AMNIAMN
Decrypted message: IAMMANN
```

Comparative Analysis of original and revised approaches:

The original Rail Fence Cipher encrypts the plaintext by writing it in a zigzag pattern across multiple rails and then reading off the characters row by row sequentially, from the first to the last rail, to generate the ciphertext. In contrast, the revised approach introduces randomness by using a random array of unique row indices to determine the order in which the rows are read during encryption. This randomizes the row selection, making the ciphertext less predictable and more secure against different cryptanalysis. However, it requires that both the sender and receiver share this specific random array to correctly decrypt the message. While the original method is simpler and more systematic, the revised approach offers enhanced security at the cost of increased complexity in key management.

Conclusion:

In conclusion, the revised Rail Fence Cipher provides improved security through randomized row selection, making it harder to decipher without the key. However, this enhancement introduces additional complexity in key management, as the random array must be securely shared between the sender and receiver.

References:

Samarth Godara, Shakti Kundu, Ravi Kaler An Improved Algorithmic Implementation of Rail Fence Cipher, 2018

Baljit Saini, Modified Ceaser Cipher and Rail fence Technique to Enhance Security, 2015

<u>Experiment 4</u> **Date: 3rd September 2024**

AIM: Study and implement a program for Columnar Transposition Cipher

<u>Original Approach</u>

Introduction:

The columnar transposition cipher is a classical encryption technique that rearranges the characters of the plaintext based on a specified keyword. It derives its name from the method of encryption, where the text is written out in rows, but the final ciphertext is read by columns.

Encryption: In the columnar transposition cipher, the plaintext is first written in rows beneath the letters of a keyword. The number of columns is determined by the length of the keyword. Once the text is arranged in this grid, the columns are then reordered according to the alphabetical order of the keyword's letters. The ciphertext is obtained by reading the characters column by column in the new order.

Decryption: To decrypt the columnar transposition cipher, create an empty grid with the number of columns equal to the length of the keyword and fill in the ciphertext column by column based on the alphabetical order of the keyword. Once the grid is filled, the original plaintext can be reconstructed by reading the text row by row.

Example:

To encrypt the word "HELLO" using a Columnar Transposition Cipher with the keyword "KEY", we first create a matrix with 3 columns (since "KEY" has 3 letters). The plaintext is written row by row, and then we rearrange the columns based on the alphabetical order of the keyword.

Keyword: KEY
Plaintext: HELLO

Matrix:
```
K E Y
--------
H E L
L O X
```
Encrypt message: EOHLLX
Read the matrix column in alphabetical order of keyword to get the ciphertext.
To decrypt the encrypted message " EOHLLX ":

1. Start by creating an empty matrix with the same number of columns as the keyword.
2. Fill in the columns with the ciphertext according to the alphabetical order of the keyword.
3. Finally, read the matrix row by row to reconstruct the plaintext.

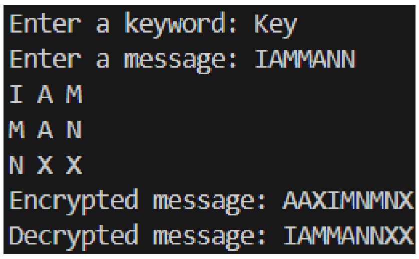
Decrypt message: "HELLOX"

Source Code:

```c
#include <stdio.h>
#include <string.h>
#include <math.h>

void removeSpace(char* message){
    char string[100];
    int k=0;

    for (int i=0; i<strlen(message); i++){
        if (message[i] != ' ') {
            string[k] = message[i];
            k++;
        }
    }
    string[k] = '\0';
    strcpy(message,string);
}

void makeMatrix(char* message, char* keyword, char
matrix[(int)ceil((double)strlen(message) /
strlen(keyword))][strlen(keyword)]){

    int msg_length = strlen(message);
    int key_length = strlen(keyword);
    int i = 0;
    int j = 0;
    int k = 0;

    for(i=0; i<(int)ceil((double)msg_length/key_length); i++){
        for(j=0; j<key_length; j++){
            if (message[k]){
                matrix[i][j] = message[k++];
            }
            else{
                matrix[i][j] = 'X';
            }
        }
    }

    for(i=0; i<(int)ceil((double)msg_length/key_length); i++){
        for(j=0; j<key_length; j++){
            printf("%c ", matrix[i][j]);
        }
        printf("\n");
    }

}
```

```c
void columnar_encrypt(char* message, char* keyword, char* encrypt_result, char
matrix[(int)ceil((double)strlen(message) /
strlen(keyword))][strlen(keyword)]){

    char result[strlen(message)];
    result[0] = '\0';
    int seen[strlen(keyword)];
    int array[strlen(keyword)];
    int k = 0;

    for(int i=0; i<strlen(keyword); i++){
        seen[i] = 0;
    }

    for (int i=0; i<strlen(keyword); i++){
        char max = 'Z';
        int temp = 0;
        for (int j=0; j<strlen(keyword); j++){
            if (max > keyword[j] && seen[j]==0){
                max = keyword[j];
                temp = j;
            }
        }
        seen[temp] = 1;
        array[k++] = temp;
    }

    for (int i=0; i<strlen(keyword); i++){
        char temp_str[2];
        for (int j=0;j<(int)ceil((double)strlen(message) /
strlen(keyword));j++){
            temp_str[0] = matrix[j][array[i]];
            temp_str[1] = '\0';
            strcat(result,temp_str);
        }
    }

    strcpy(encrypt_result,result);

}

void columnar_decrypt(char* encrypt_message, char* keyword, char*
decrypt_result){

    char result[strlen(encrypt_message)];
    result[0] = '\0';
    int seen[strlen(keyword)];
    int array[strlen(keyword)];
    char matrix[(int)ceil((double)strlen(encrypt_message) /
strlen(keyword))][strlen(keyword)];
    int k = 0;

    for(int i=0; i<strlen(keyword); i++){
        seen[i] = 0;
    }

    for (int i=0; i<strlen(keyword); i++){
        char max = 'Z';
        int temp = 0;
        for (int j=0; j<strlen(keyword); j++){
            if (max > keyword[j] && seen[j]==0){
                max = keyword[j];
                temp = j;
            }
        }
        seen[temp] = 1;
        array[k++] = temp;
    }
    k=0;
    for (int i=0; i<strlen(keyword); i++){
        char temp_str[2];
        for (int j=0;j<(int)ceil((double)strlen(encrypt_message) /
strlen(keyword));j++){
            matrix[j][array[i]] = encrypt_message[k++];
        }
    }

    for (int i=0; i<(int)ceil((double)strlen(encrypt_message) /
strlen(keyword)); i++){
        for (int j=0; j<strlen(keyword); j++){
            char temp_str[2];
            temp_str[0] = matrix[i][j];
            temp_str[1] = '\0';
            strcat(result,temp_str);
        }
    }

    strcpy(decrypt_result,result);

}

void main(){

    char keyword[100];
    char message[100];
    printf("Enter a keyword: ");
    gets(keyword);
    strupr(keyword);

    printf("Enter a message: ");
    gets(message);
    strupr(message);
    removeSpace(message);

    char matrix[(int)ceil((double)strlen(message) /
strlen(keyword))][strlen(keyword)];
    char encrypt_message[strlen(message)];
    char decrypt_message[strlen(message)];

    makeMatrix(message, keyword, matrix);

    columnar_encrypt(message, keyword, encrypt_message, matrix);

    printf("Encrypted message: ");
    puts(encrypt_message);

    columnar_decrypt(encrypt_message, keyword, decrypt_message);

    printf("Decrypted message: ");
    puts(decrypt_message);

}
```

Output screenshot:

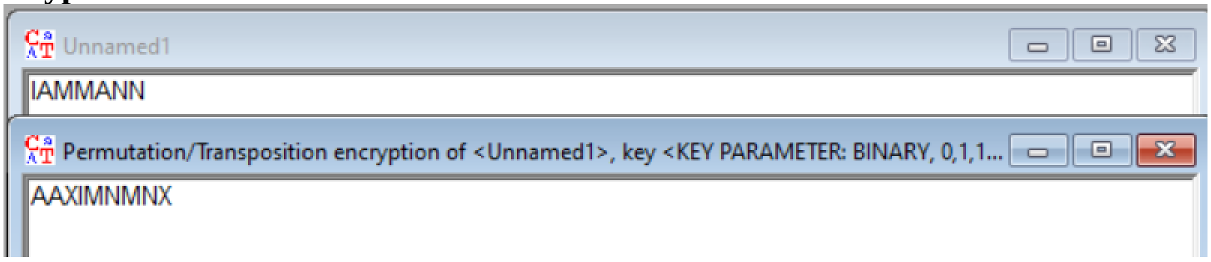


```
Enter a keyword: Key
Enter a message: IAMMANN
I A M
M A N
N X X
Encrypted message: AAXIMNMNX
Decrypted message: IAMMANNXX
```

Cryptanalysis:

Cryptanalysis of the Columnar Transposition Cipher involves several techniques to break the cipher and uncover the plaintext without knowing the key. One method is using known plaintext, where if a portion of the original message is known, it can be aligned with the ciphertext to deduce the column order by analysing patterns. Another approach is brute force, where all possible permutations of column orders are tested; this method is feasible for short keys but becomes impractical with longer keys due to the exponential increase in possible combinations. Additionally, analysing transposition frequencies by matching common patterns or structures in the ciphertext with typical plaintext arrangements can also help in deciphering the message.

Cryptool screenshot:



IAMMANN

Permutation/Transposition encryption of <Unnamed1>, key <KEY PARAMETER: BINARY, 0,1,1...

AAXIMNMNX

Revised Approach

Introduction:
In the revised approach, the key is first encrypted using a Caesar cipher, and the resulting encrypted key is then used to determine the column order in the matrix for the columnar transposition cipher. This added layer of encryption enhances security by obscuring the original key, making it more difficult for attackers to deduce the correct column order even if they manage to analyse the transposition patterns.

Example:
To encrypt the word "HELLO" using a Columnar Transposition Cipher with the keyword "KEY," we first encrypt the keyword using a Caesar cipher with a shift of 3. The Caesar cipher transforms "KEY" into "NHB." Next, we create a matrix with 3 columns (since "NHB" has 3 letters). The plaintext is written row by row, and the columns are rearranged according to the alphabetical order of the encrypted keyword.

Keyword: KEY → NHB
Plaintext: HELLO

N H B

H E L
L O X

Encrypt message: "LXEOHL"

To decrypt the encrypted message " LXEOHL ":

1. Start by creating an empty matrix with the same number of columns as the encrypted keyword.
2. Fill in the columns with the ciphertext according to the alphabetical order of the keyword "NHB."
3. Finally, read the matrix row by row to reconstruct the plaintext.

Decrypt message: "HELLOX"

Source Code:

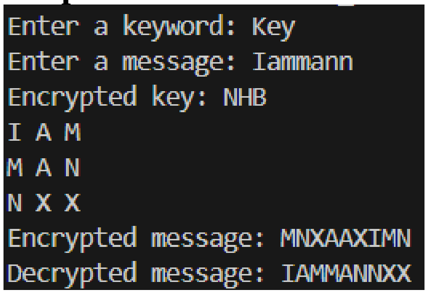
```
#include "info_sec.h"

#include <stdio.h>
```

```c
#include <string.h>
#include <math.h>

void removeSpace(char* message){
    char string[100];
    int k=0;

    for (int i=0; i<strlen(message); i++){
        if (message[i] != ' ') {
            string[k] = message[i];
            k++;
        }
    }
    string[k] = '\0';
    strcpy(message,string);
}

void makeMatrix(char* message, char* keyword, char
matrix[(int)ceil((double)strlen(message) /
strlen(keyword))][strlen(keyword)]){

    int msg_length = strlen(message);
    int key_length = strlen(keyword);
    int i = 0;
    int j = 0;
    int k = 0;

    for(i=0; i<(int)ceil((double)msg_length/key_length); i++){
        for(j=0; j<key_length; j++){
            if (message[k]){
                matrix[i][j] = message[k++];
            }
            else{
                matrix[i][j] = 'X';
            }
        }
    }

    for(i=0; i<(int)ceil((double)msg_length/key_length); i++){
        for(j=0; j<key_length; j++){
            printf("%c ", matrix[i][j]);
        }
        printf("\n");
    }

}
```

```
void columnar_encrypt(char* message, char* keyword, char* encrypt_result, char
matrix[(int)ceil((double)strlen(message) /
strlen(keyword))][strlen(keyword)]){

    char result[strlen(message)];
    result[0] = '\0';
    int seen[strlen(keyword)];
    int array[strlen(keyword)];
    int k = 0;

    for(int i=0; i<strlen(keyword); i++){
        seen[i] = 0;
    }

    for (int i=0; i<strlen(keyword); i++){
        char max = 'Z';
        int temp = 0;
        for (int j=0; j<strlen(keyword); j++){
            if (max > keyword[j] && seen[j]==0){
                max = keyword[j];
                temp = j;
            }
        }
        seen[temp] = 1;
        array[k++] = temp;
    }

    for (int i=0; i<strlen(keyword); i++){
        char temp_str[2];
        for (int j=0;j<(int)ceil((double)strlen(message) /
strlen(keyword));j++){
            temp_str[0] = matrix[j][array[i]];
            temp_str[1] = '\0';
            strcat(result,temp_str);
        }
    }

    strcpy(encrypt_result,result);

}

void columnar_decrypt(char* encrypt_message, char* keyword, char*
decrypt_result){

    char result[strlen(encrypt_message)];
    result[0] = '\0';
    int seen[strlen(keyword)];
    int array[strlen(keyword)];
    char matrix[(int)ceil((double)strlen(encrypt_message) / 
strlen(keyword))][strlen(keyword)];
    int k = 0;

    for(int i=0; i<strlen(keyword); i++){
        seen[i] = 0;
    }

    for (int i=0; i<strlen(keyword); i++){
        char max = 'Z';
        int temp = 0;
        for (int j=0; j<strlen(keyword); j++){
            if (max > keyword[j] && seen[j]==0){
                max = keyword[j];
                temp = j;
            }
        }
        seen[temp] = 1;
        array[k++] = temp;
    }
    k=0;
    for (int i=0; i<strlen(keyword); i++){
        char temp_str[2];
        for (int j=0;j<(int)ceil((double)strlen(encrypt_message) / 
strlen(keyword));j++){
            matrix[j][array[i]] = encrypt_message[k++];
        }
    }

    for (int i=0; i<(int)ceil((double)strlen(encrypt_message) / 
strlen(keyword)); i++){
        for (int j=0; j<strlen(keyword); j++){
            char temp_str[2];
            temp_str[0] = matrix[i][j];
            temp_str[1] = '\0';
            strcat(result,temp_str);
        }
    }

    strcpy(decrypt_result,result);

}

void main(){

    char keyword[100];
    char message[100];
    char encrypt_key[100];
```

```c
    char alphabets[26] = {'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'};

    printf("Enter a keyword: ");
    gets(keyword);
    strupr(keyword);

    encrypt(keyword,alphabets,encrypt_key);

    printf("Enter a message: ");
    gets(message);
    strupr(message);
    removeSpace(message);

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

    char matrix[(int)ceil((double)strlen(message) /
strlen(encrypt_key))][strlen(encrypt_key)];
    char encrypt_message[strlen(message)];
    char decrypt_message[strlen(message)];

    makeMatrix(message, encrypt_key, matrix);

    columnar_encrypt(message, encrypt_key, encrypt_message, matrix);

    printf("Encrypted message: ");
    puts(encrypt_message);

    columnar_decrypt(encrypt_message, encrypt_key, decrypt_message);

    printf("Decrypted message: ");
    puts(decrypt_message);

}
```

Output screenshot:



```
Enter a keyword: Key
Enter a message: Iammann
Encrypted key: NHB
I A M
M A N
N X X
Encrypted message: MNXAAXIMN
Decrypted message: IAMMANNXX
```

Comparative Analysis of original and revised approaches:

The original columnar transposition cipher encrypts data based on a keyword, which determines the arrangement of columns for transposition. However, this method is susceptible to cryptanalysis techniques that can reveal the keyword, compromising the encryption. In contrast, the revised approach enhances security by applying a Caesar cipher to the keyword before using it for columnar transposition. This additional step transforms the keyword by shifting its letters a fixed number of positions in the alphabet, thus increasing the complexity of the cipher. By masking the original keyword, the revised method makes it significantly more difficult for attackers to deduce the keyword or decipher the message. This added layer of encryption not only strengthens the overall security of the cipher but also improves resistance to frequency analysis and other cryptographic attacks, making the encrypted message more secure and less vulnerable to exposure.

Conclusion:

In conclusion, the revised columnar transposition cipher, which incorporates a Caesar cipher for keyword transformation, offers enhanced security. This added complexity makes it more difficult for attackers to deduce the keyword, strengthening the overall encryption and improving resistance to cryptographic attacks.

References:

Dian Rachmawati, Sri Melvani Hardi, Raju Partogi Combination of columnar transposition cipher caesar cipher – April 2019
Malay B. Pramanik Implementation of Cryptography Technique using Columnar Transposition – Jan 2014

Experiment 5 **Date: 10th September 2024**

AIM: Study and implement a program for Vigenère Cipher

Original Approach

Introduction:

The Vigenère Cipher is a type of substitution technique that uses a keyword to systematically alter the plaintext. To encrypt a message, you first select a keyword and extend it to match the length of the plaintext by repeating it. Each letter in the plaintext is then shifted according to the corresponding letter in the keyword, where the position of the letter in the alphabet determines the shift amount. This method generates a ciphertext where each letter is encrypted based on its position and the keyword.

Example:

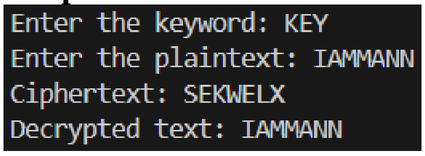
To encrypt the text "HELLO WORLD" using the Vigenère Cipher with the keyword "KEY," we begin by repeating the keyword to match the length of the plaintext, resulting in "KEYKEYKEYKE." Next, we shift each letter of the plaintext according to the position of the corresponding letter in the repeated keyword. For example, the letter 'h' in "HELLO" is shifted by 10 positions (the value of 'k') to become 'r', 'e' is shifted by 4 (the value of 'e') to become 'i', and so on. Continuing this process, "HELLO" becomes "RIJVS." The space remains unchanged. For the second part, "WORLD," we apply the shifts in the same way: 'w' is shifted by 24 (the value of 'y') to become 'u', 'o' by 10 (the value of 'k') to become 'y', and so forth, transforming "WORLD" into "UYVJN." Thus, the complete ciphertext for "HELLO WORLD" with the keyword "KEY" is " RIJVS UYVJN"

Source Code:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void vigenereCipherEncrypt(char plaintext[], char encrypttext[], char
keyword[]) {
    int textLength = strlen(plaintext);
    int keywordLength = strlen(keyword);
    char ciphertext[textLength + 1];

    for (int i = 0; i < keywordLength; i++) {
        keyword[i] = tolower(keyword[i]);
    }

    for (int i = 0, j = 0; i < textLength; i++) {
        char currentChar = plaintext[i];

        if (isalpha(currentChar)) {
            char keyChar = keyword[j % keywordLength];
            keyChar = tolower(keyChar);
            if (islower(currentChar)) {
                ciphertext[i] = ((currentChar - 'a') + (keyChar - 'a')) % 26 +
'a';
            } else if (isupper(currentChar)) {
                ciphertext[i] = ((currentChar - 'A') + (keyChar - 'a')) % 26 +
'A';
            }

            j++;
        } else {
            ciphertext[i] = currentChar;
        }
    }

    ciphertext[textLength] = '\0';
    printf("Ciphertext: %s\n", ciphertext);
    strcpy(encrypttext,ciphertext);
}

void vigenereCipherDecrypt(char ciphertext[], char decrypttext[], char
keyword[]) {
    int textLength = strlen(ciphertext);
    int keywordLength = strlen(keyword);
    char plaintext[textLength + 1];

    for (int i = 0; i < keywordLength; i++) {
        keyword[i] = tolower(keyword[i]);
    }

    for (int i = 0, j = 0; i < textLength; i++) {
        char currentChar = ciphertext[i];

        if (isalpha(currentChar)) {
            char keyChar = keyword[j % keywordLength];
            keyChar = tolower(keyChar);

            if (islower(currentChar)) {
                plaintext[i] = ((currentChar - 'a') - (keyChar - 'a') + 26) %
26 + 'a';
            } else if (isupper(currentChar)) {
                plaintext[i] = ((currentChar - 'A') - (keyChar - 'a') + 26) %
26 + 'A';
            }

            j++;
        } else {
            plaintext[i] = currentChar;
        }
```

```c
    }

    plaintext[textLength] = '\0';
    printf("Decrypted text: %s\n", plaintext);
    strcpy(decrypttext,plaintext);
}

int main() {
    char plaintext[100], encrypttext[100], decrypttext[100], keyword[100];
    int choice;

    printf("Enter the keyword: ");
    fgets(keyword, sizeof(keyword), stdin);
    keyword[strcspn(keyword, "\n")] = 0;

    printf("Enter the plaintext: ");
    gets(plaintext);
    vigenereCipherEncrypt(plaintext, encrypttext, keyword);
    vigenereCipherDecrypt(encrypttext, decrypttext, keyword);

    return 0;
}
```

Output screenshot:

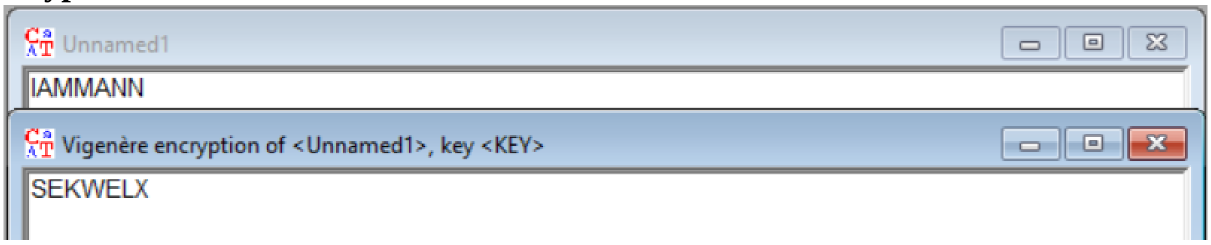


```
Enter the keyword: KEY
Enter the plaintext: IAMMANN
Ciphertext: SEKWELX
Decrypted text: IAMMANN
```

Cryptanalysis:

In a known plaintext attack, the attacker has access to a portion of both the plaintext and its corresponding ciphertext. Since the Vigenère cipher repeats its key, the attacker can use the known plaintext to reverse the encryption process and directly uncover the key. Once part of the key is revealed, the attacker can decrypt other parts of the ciphertext encrypted with the same key, because the key is cyclic.

Cryptool screenshot:



Unnamed1

IAMMANN

Vigenère encryption of <Unnamed1>, key <KEY>

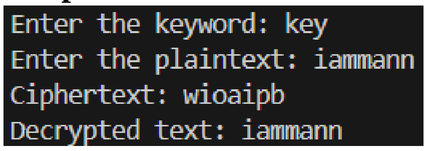
SEKWELX

Revised Approach

Introduction:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void vigenereCipherEncrypt(char plaintext[], char encrypttext[], char
keyword[]) {
    int textLength = strlen(plaintext);
    int keywordLength = strlen(keyword);
    char ciphertext[textLength + 1];
    int ldiff = textLength - keywordLength;

    for (int i = 0; i < keywordLength; i++) {
        keyword[i] = tolower(keyword[i]);
    }

    for (int i = 0, j = 0; i < textLength; i++) {
        char currentChar = tolower(plaintext[i]);

        if (isalpha(currentChar)) {
            char keyChar = keyword[j % keywordLength];
            ciphertext[i] = ((currentChar - 'a') + (keyChar - 'a') + ldiff) %
26 + 'a';
            j++;
        } else {
            ciphertext[i] = currentChar;
        }
    }

    ciphertext[textLength] = '\0';
    printf("Ciphertext: %s\n", ciphertext);
    strcpy(encrypttext, ciphertext);
}

void vigenereCipherDecrypt(char ciphertext[], char decrypttext[], char
keyword[]) {
    int textLength = strlen(ciphertext);
    int keywordLength = strlen(keyword);
    char plaintext[textLength + 1];
    int ldiff = textLength - keywordLength;

    for (int i = 0; i < keywordLength; i++) {
        keyword[i] = tolower(keyword[i]);
    }

    for (int i = 0, j = 0; i < textLength; i++) {
        char currentChar = tolower(ciphertext[i]);

        if (isalpha(currentChar)) {
            char keyChar = keyword[j % keywordLength];
            plaintext[i] = ((currentChar - 'a') - (keyChar - 'a') - ldiff +
26) % 26 + 'a';
            j++;
        } else {
            plaintext[i] = currentChar;
        }
    }

    plaintext[textLength] = '\0';
    printf("Decrypted text: %s\n", plaintext);
    strcpy(decrypttext, plaintext);
}

int main() {
    char plaintext[100], encrypttext[100], decrypttext[100], keyword[100];

    printf("Enter the keyword: ");
    fgets(keyword, sizeof(keyword), stdin);
    keyword[strcspn(keyword, "\n")] = 0;

    printf("Enter the plaintext: ");
    fgets(plaintext, sizeof(plaintext), stdin);
    plaintext[strcspn(plaintext, "\n")] = 0;

    vigenereCipherEncrypt(plaintext, encrypttext, keyword);
    vigenereCipherDecrypt(encrypttext, decrypttext, keyword);

    return 0;
}
```

Output screenshot:



```
Enter the keyword: key
Enter the plaintext: iammann
Ciphertext: wioaipb
Decrypted text: iammann
```

<u>Comparative Analysis of original and revised approaches:</u>

The original approach to the Vigenère cipher introduces a modification by incorporating a difference value, `ldiff`, calculated from the lengths of the plaintext and keyword. This results in adding `ldiff` to the shift calculation during encryption and subtracting it during decryption, which deviates from the traditional Vigenère cipher method. This adjustment introduces additional complexity and variability, making the encryption and decryption processes less straightforward and potentially harder to decode accurately. In contrast, the revised approach adheres to the conventional Vigenère cipher methodology, where encryption and decryption rely solely on the keyword for shifting characters. By removing `ldiff`, the revised version simplifies the process, ensuring that the cipher operates as intended and making it easier to decode accurately. This traditional approach promotes consistency and predictability in the encryption and decryption outcomes, aligning with the standard Vigenère cipher principles.

Conclusion:

In conclusion, the revised approach to the Vigenère cipher adheres more closely to traditional methods by focusing solely on the keyword for character shifts during encryption and decryption. This makes the cipher simpler, more predictable, and easier to implement accurately compared to the original approach, which introduces additional complexity. The revised method provides a more standard and reliable implementation of the Vigenère cipher, ensuring consistency in encryption and decryption processes.

References:

William Stallings - Cryptography and Network Security Principles and Practice, 2010
Aiman Al-Sabaawi - Cryptanalysis of Vigenère Cipher: Method Implementation, 2020

Experiment 6 **Date:**

AIM: Study and implement a program for n-gram Hill Cipher

Original Approach

Introduction: The Hill cipher is a polygraphic substitution cipher based on linear algebra. It was invented by Lester S. Hill in 1929 and uses matrix multiplication to transform plaintext blocks into ciphertext. The cipher works by dividing the plaintext into fixed-size blocks and encrypting them using an invertible key matrix. It provides strong security for its time, leveraging linear transformations to create more complex and secure ciphertext compared to simpler classical ciphers. However, its security relies on the key matrix being invertible in modulo arithmetic, which makes key choice crucial.

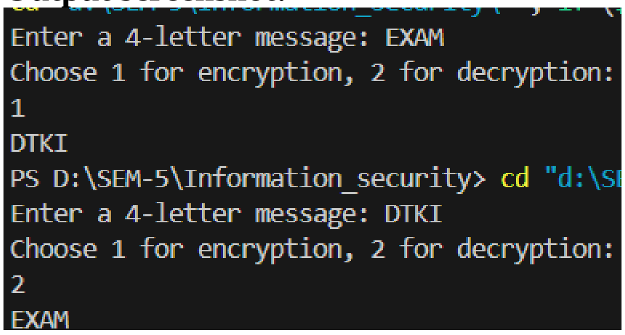
Example:
In the Hill cipher encryption process for the plaintext "exam" and the key "hill," we first convert the plaintext into pairs of letters, "ex" and "am." These are then transformed into numerical values based on their positions in the alphabet, where 'e' = 4, 'x' = 23, 'a' = 0, and 'm' = 12, forming the matrices [[4], [23]] and [[0], [12]]. The key "hill" is similarly converted into a 2x2 matrix based on the letters' positions: [[7, 8], [11, 11]], where 'h' = 7, 'i' = 8, 'l' = 11. To encrypt, we multiply the key matrix by each plaintext matrix and reduce the results modulo 26. The encryption produces the ciphertext matrices [[4], [11]] and [[18], [2]], which correspond to the letters "el" and "sc." For decryption, we calculate the inverse of the key matrix mod 26 and use it to retrieve the original plaintext from the ciphertext.

Source Code:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define SIZE 2

// Function to find the modular inverse of a number 'a' mod m
int modInverse(int a, int m) {
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1)
            return x;
    }
    return -1;  // If inverse doesn't exist
}

// Function to calculate the determinant of a 2x2 matrix
int determinant(int key[SIZE][SIZE]) {
    return (key[0][0] * key[1][1] - key[0][1] * key[1][0]);
}

// Function to find the inverse of a 2x2 matrix mod 26
void inverseKeyMatrix(int key[SIZE][SIZE], int inverseKey[SIZE][SIZE]) {
    int det = determinant(key);
    int detInverse = modInverse(det % 26, 26);  // Find modular inverse of
determinant mod 26

    if (detInverse == -1) {
        printf("Inverse doesn't exist!\n");
        return;
    }

    // Calculate inverse key matrix mod 26
    inverseKey[0][0] = (key[1][1] * detInverse) % 26;
    inverseKey[1][1] = (key[0][0] * detInverse) % 26;
    inverseKey[0][1] = (-key[0][1] * detInverse) % 26;
    inverseKey[1][0] = (-key[1][0] * detInverse) % 26;

    // Make sure all elements are positive by adding 26 if any element is
negative
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (inverseKey[i][j] < 0)
                inverseKey[i][j] += 26;
        }
    }
}

// Encryption function
void encrypt(char message[], int key[SIZE][SIZE]) {
    int messageVector[SIZE];
    int cipherVector[SIZE];
    int i, j, k = 0;

    // Making matrix of 2
    while (k < strlen(message)) {
        // Prepare message vector for the current block
        for (i = 0; i < SIZE; i++) {
            if (isalpha(message[k])) {
                messageVector[i] = toupper(message[k]) - 'A';  // Convert
letter to number
            }
            k++;
        }

        // Encrypt the block using matrix multiplication
        for (i = 0; i < SIZE; i++) {
            cipherVector[i] = 0;
            for (j = 0; j < SIZE; j++) {
                cipherVector[i] += key[i][j] * messageVector[j];
            }
```

```c
            cipherVector[i] %= 26;  // Mod 26 for each character
        }

        // Output the encrypted block
        for (i = 0; i < SIZE; i++) {
            printf("%c", cipherVector[i] + 'A');  // Convert number back to
letter
        }

    }
}

// Decryption function
void decrypt(char cipher[], int key[SIZE][SIZE]) {
    int cipherVector[SIZE];
    int messageVector[SIZE];
    int inverseKey[SIZE][SIZE];
    int i, j, k = 0;

    // Find the inverse of the key matrix
    inverseKeyMatrix(key, inverseKey);

    // Making matrix of 2
    while (k < strlen(cipher)) {
        // Prepare cipher vector for the current block
        for (i = 0; i < SIZE; i++) {
            if (isalpha(cipher[k])) {
                cipherVector[i] = toupper(cipher[k]) - 'A';  // Convert letter
to number
            }
            k++;
        }

        // Decrypt the block using matrix multiplication with inverse key
matrix
        for (i = 0; i < SIZE; i++) {
            messageVector[i] = 0;
            for (j = 0; j < SIZE; j++) {
                messageVector[i] += inverseKey[i][j] * cipherVector[j];
            }
            messageVector[i] %= 26;  // Mod 26 for each character

            // Adjust for negative values
            if (messageVector[i] < 0)
                messageVector[i] += 26;
        }

        // Output the decrypted block
        for (i = 0; i < SIZE; i++) {
            printf("%c", messageVector[i] + 'A');  // Convert number back to
letter
        }
    }
}

int main() {
    char message[5];  // 4 characters for the 2x2 Hill cipher (with space for
null terminator)
    int key[SIZE][SIZE] = {{3, 3}, {2, 5}};  // Example key matrix
    int choice;

    printf("Enter a 4-letter message: ");
    scanf("%4s", message);  // Limit input to 4 characters

    printf("Choose 1 for encryption, 2 for decryption:\n");
    scanf("%d",&choice);

    switch (choice)
    {
    case 1:
        encrypt(message,key);
        break;

    case 2:
        decrypt(message,key);

    default:
        break;
    }

    return 0;
}
```

Output screenshot:



```
Enter a 4-letter message: EXAM
Choose 1 for encryption, 2 for decryption:
1
DTKI
PS D:\SEM-5\Information_security> cd "d:\SE
Enter a 4-letter message: DTKI
Choose 1 for encryption, 2 for decryption:
2
EXAM
```

Cryptanalysis:

The original approach of this Hill cipher implementation uses a basic 2x2 key matrix to encrypt and decrypt a 4-letter message through matrix multiplication, where the message is transformed into a vector, multiplied by the key matrix (mod 26), and converted back into text. The revised approach, which adds the key's first column to the ciphertext's first row and the second column to the second row, fundamentally alters the encryption process, potentially creating a more complex relationship between the ciphertext and plaintext. However, this modification may introduce predictable patterns in the ciphertext, increasing the risk of vulnerability if a cryptanalyst can exploit these consistencies, especially if partial plaintext-ciphertext pairs are known. The security of the revised cipher depends on whether this key modification provides sufficient diffusion and avoids generating a linear structure that could simplify matrix inversion during cryptanalysis.

Cryptool screenshot:

Revised Approach

Introduction:
The Hill cipher is a classical encryption method that uses linear algebra to secure messages by transforming blocks of plaintext into ciphertext through matrix multiplication. In the traditional 2x2 Hill cipher, a key matrix is employed to scramble pairs of letters, enhancing security compared to simple substitution ciphers. This revised approach modifies the encryption process by adding the columns of the key matrix directly to the rows of the resulting ciphertext. This adjustment aims to create a more intricate relationship between the key and the ciphertext, enhancing the cipher's complexity and resistance to cryptanalysis while still adhering to the foundational principles of the Hill cipher.

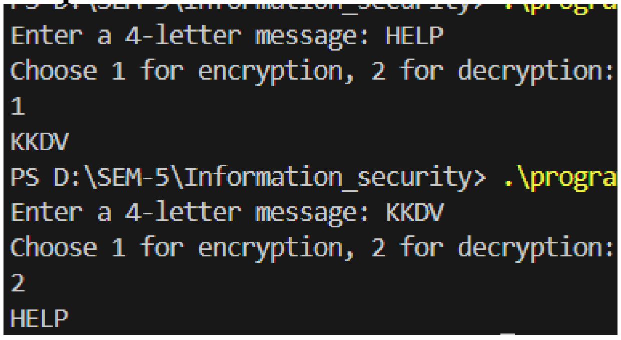
Source Code:

```c
#include <stdio.h>

#include <string.h>
#include <ctype.h>

#define SIZE 2

// Function to find the modular inverse of a number 'a' mod m
int modInverse(int a, int m) {
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1)
            return x;
    }
    return -1;  // If inverse doesn't exist
}

// Function to calculate the determinant of a 2x2 matrix
int determinant(int key[SIZE][SIZE]) {
    return (key[0][0] * key[1][1] - key[0][1] * key[1][0]);
}

// Function to find the inverse of a 2x2 matrix mod 26
```

```c
void inverseKeyMatrix(int key[SIZE][SIZE], int inverseKey[SIZE][SIZE]) {
    int det = determinant(key);
    int detInverse = modInverse(det % 26, 26);  // Find modular inverse of
determinant mod 26

    if (detInverse == -1) {
        printf("Inverse doesn't exist!\n");
        return;
    }

    // Calculate inverse key matrix mod 26
    inverseKey[0][0] = (key[1][1] * detInverse) % 26;
    inverseKey[1][1] = (key[0][0] * detInverse) % 26;
    inverseKey[0][1] = (-key[0][1] * detInverse) % 26;
    inverseKey[1][0] = (-key[1][0] * detInverse) % 26;

    // Make sure all elements are positive by adding 26 if any element is
negative
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (inverseKey[i][j] < 0)
                inverseKey[i][j] += 26;
        }
    }
}

// Encryption function
void encrypt(char message[], int key[SIZE][SIZE]) {
    int messageVector[SIZE];
    int cipherVector[SIZE];
    int i, j, k = 0;
    int keyvector[SIZE] = {key[0][0],key[1][0],key[1][0],key[1][1]};

    // Making matrix of 2
    while (k < strlen(message)) {
        // Prepare message vector for the current block
        for (i = 0; i < SIZE; i++) {
            if (isalpha(message[k])) {
                messageVector[i] = toupper(message[k]) - 'A';  // Convert
letter to number
            }
            k++;
        }

        // Encrypt the block using matrix multiplication
        for (i = 0; i < SIZE; i++) {
            cipherVector[i] = 0;
            for (j = 0; j < SIZE; j++) {
                cipherVector[i] += key[i][j] * messageVector[j] ;
            }
            cipherVector[i] += keyvector[i];
            cipherVector[i] %= 26;  // Mod 26 for each character
        }

        // Output the encrypted block
        for (i = 0; i < SIZE; i++) {
            printf("%c", cipherVector[i] + 'A');  // Convert number back to
letter
        }

    }
}

// Decryption function
void decrypt(char cipher[], int key[SIZE][SIZE]) {
    int cipherVector[SIZE];
    int messageVector[SIZE];
    int inverseKey[SIZE][SIZE];
    int keyvector[SIZE] = {key[0][0],key[1][0],key[1][0],key[1][1]};
    int i, j, k = 0;

    // Find the inverse of the key matrix
    inverseKeyMatrix(key, inverseKey);

    // Making matrix of 2
    while (k < strlen(cipher)) {
        // Prepare cipher vector for the current block
        for (i = 0; i < SIZE; i++) {
            if (isalpha(cipher[k])) {
                cipherVector[i] = toupper(cipher[k]) - 'A';  // Convert letter
to number
                // Subtract the keyvector value (undoing the addition during
encryption)
                cipherVector[i] = (cipherVector[i] - keyvector[i] + 26) % 26;
            }
            k++;
        }

        // Decrypt the block using matrix multiplication with inverse key
matrix
        for (i = 0; i < SIZE; i++) {
            messageVector[i] = 0;
            for (j = 0; j < SIZE; j++) {
                messageVector[i] += inverseKey[i][j] * cipherVector[j];
            }
            messageVector[i] %= 26;  // Mod 26 for each character
```

```c
            // Adjust for negative values
            if (messageVector[i] < 0)
                messageVector[i] += 26;
        }

        // Output the decrypted block
        for (i = 0; i < SIZE; i++) {
            printf("%c", messageVector[i] + 'A');  // Convert number back to
letter
        }
    }
}

int main() {
    char message[5];  // 4 characters for the 2x2 Hill cipher (with space for
null terminator)
    int key[SIZE][SIZE] = {{3, 3}, {2, 5}};  // Example key matrix
    int choice;

    printf("Enter a 4-letter message: ");
    scanf("%4s", message);  // Limit input to 4 characters

    printf("Choose 1 for encryption, 2 for decryption:\n");
    scanf("%d",&choice);

    switch (choice)
    {
    case 1:
        encrypt(message,key);
        break;

    case 2:
        decrypt(message,key);

    default:
        break;
    }

    return 0;
}
```

Output Screenshot:



```
PS D:\SEM-5\Information_security> .\progra
Enter a 4-letter message: HELP
Choose 1 for encryption, 2 for decryption:
1
KKDV
PS D:\SEM-5\Information_security> .\progra
Enter a 4-letter message: KKDV
Choose 1 for encryption, 2 for decryption:
2
HELP
```

Conclusion:

In conclusion, the original Hill cipher approach utilizes a 2x2 key matrix to transform pairs of plaintext characters into ciphertext through matrix multiplication, providing effective security against frequency analysis. The revised approach enhances this method by adding the columns of the key matrix to the ciphertext rows, creating a more complex relationship that aims to improve resistance to cryptanalysis. While both methods illustrate the importance of key management and the underlying mathematical principles of encryption, the revised approach seeks to increase diffusion and strengthen security through its modifications.

References:

G. Rekha, V. Srinavas - A Novel Approach in Hill Cipher Cryptography, 2023
K. Adinarayana Reddy, B. Vishnuvardhan, Madhuviswanatham, A.V.N. Krishna - A Modified Hill Cipher Based on Circulant Matrices, 2012

Experiment 7   Date:

AIM: Study and Use of RSA algorithm (encryption and decryption)

Original Approach

Introduction: RSA (Rivest-Shamir-Adleman) is a widely used public-key cryptographic system that enables secure data transmission and digital signatures. Developed in 1977, RSA relies on the mathematical properties of large prime numbers. It involves two keys: a public key for encryption, which can be shared openly, and a private key for decryption, kept secret by the owner. The security of RSA hinges on the difficulty of factoring the product of two large prime numbers, making it computationally infeasible for attackers to derive the private key from the public key. This makes RSA a cornerstone of modern internet security, facilitating secure communications in various applications, including web browsing, email, and digital signatures.

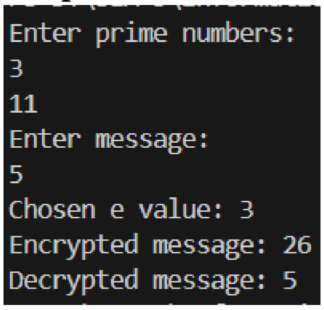
Example:

Choose p = 3 and q = 11
Compute n = p * q = 3 * 11 = 33
Compute φ(n) = (p - 1) * (q - 1) = 2 * 10 = 20
Choose e such that 1 < e < φ(n) and e and φ (n) are coprime. Let e = 7
Compute a value for d such that (d * e) % φ(n) = 1. One solution is d = 3 [(3 * 7) % 20 = 1]
Public key is (e, n) => (7, 33)
Private key is (d, n) => (3, 33)
The encryption of $m = 2$ is $c = 2^7 \% 33 = 29$
The decryption of $c = 29$ is $m = 29^3 \% 33 = 2$

Source Code:

```c
#include <stdio.h>
#include <string.h>

long int gcd(long int a, long int b) {
    while (b != 0) {
        long int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

long int mod_exp(long int base, long int exponent, long int mod) {
    long int result = 1;
    base = base % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exponent = exponent / 2;
    }
    return result;
}

long int encrypt(long int m, long int e, long int n) {
    return mod_exp(m, e, n);
}

long int decrypt(long int c, long int e, long int n, long int totient_n) {
    long int d = 1;
    while ((e * d) % totient_n != 1) {
        d++;
    }
    return mod_exp(c, d, n);
}

void main() {
    long int p, q, m, n, totient_n, e;
    long int encrypt_m, decrypt_m;

    printf("Enter prime numbers: \n");
    scanf("%ld", &p);
    scanf("%ld", &q);
    printf("Enter message: \n");
    scanf("%ld", &m);

    n = p * q;
    totient_n = (p - 1) * (q - 1);

    for (long int i = 2; i < totient_n; i++) {
        if (gcd(i, totient_n) == 1) {
            e = i;
            break;
        }
    }

    printf("Chosen e value: %ld\n", e);

    encrypt_m = encrypt(m, e, n);
    printf("Encrypted message: %ld\n", encrypt_m);

    decrypt_m = decrypt(encrypt_m, e, n, totient_n);
    printf("Decrypted message: %ld\n", decrypt_m);
}
```

Output Screenshot:



```
Enter prime numbers:
3
11
Enter message:
5
Chosen e value: 3
Encrypted message: 26
Decrypted message: 5
```

Cryptanalysis:

Cryptanalysis of RSA primarily targets the security weaknesses stemming from its reliance on the difficulty of factoring large composite numbers. The most significant attack is factorization, where an attacker seeks to decompose the product of two large primes (the modulus \(n = p \times q \)) into its constituent primes, using advanced algorithms like the General Number Field Sieve. Other vulnerabilities include common modulus attacks, where multiple users share the same modulus but different public exponents, and low exponent attacks, which exploit small public exponents (like 3) when the same message is encrypted for different recipients. Additionally, side-channel attacks can analyze timing information or power consumption during decryption operations to glean information about the private key. While RSA remains secure when implemented with sufficiently large keys and proper precautions, awareness of these potential vulnerabilities is crucial for maintaining robust cryptographic systems.