Program 37 C program for Hill cipher succumbs to a known plaintext

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <math.h>

#define MAX\_SIZE 5

// Function to encrypt a message using the Hill cipher

void hillCipherEncrypt(const int key[MAX\_SIZE][MAX\_SIZE], const char \*plaintext, char \*ciphertext, int n) {

int len = strlen(plaintext);

int padded\_len = (int)ceil((double)len / n) \* n;

// Allocate memory for the padded plaintext

char \*padded\_text = (char \*)malloc(padded\_len + 1);

strncpy(padded\_text, plaintext, len);

// Pad the plaintext with 'X' if needed

for (int i = len; i < padded\_len; ++i) {

padded\_text[i] = 'X';

}

padded\_text[padded\_len] = '\0';

// Perform encryption

for (int i = 0; i < padded\_len; i += n) {

for (int j = 0; j < n; ++j) {

int sum = 0;

for (int k = 0; k < n; ++k) {

sum += key[j][k] \* (padded\_text[i + k] - 'A');

}

ciphertext[i + j] = (sum % 26) + 'A';

}

}

free(padded\_text);

}

// Function to decrypt a message using the Hill cipher

void hillCipherDecrypt(const int key[MAX\_SIZE][MAX\_SIZE], const char \*ciphertext, char \*decrypted\_text, int n) {

int len = strlen(ciphertext);

// Perform decryption

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

for (int j = 0; j < n; ++j) {

int sum = 0;

for (int k = 0; k < n; ++k) {

sum += key[j][k] \* (ciphertext[i + k] - 'A');

}

decrypted\_text[i + j] = (sum % 26 + 26) % 26 + 'A';

}

}

}

// Function to find the modular multiplicative inverse

int modInverse(int a, int m) {

a = a % m;

for (int x = 1; x < m; x++) {

if ((a \* x) % m == 1) {

return x;

}

}

return -1; // No modular inverse exists

}

// Function to perform a known-plaintext attack on the Hill cipher

bool knownPlaintextAttack(const char \*plaintext, const char \*ciphertext, int key[MAX\_SIZE][MAX\_SIZE], int n) {

int len = strlen(plaintext);

int mat[MAX\_SIZE][MAX\_SIZE];

// Construct the matrix from known plaintext-ciphertext pairs

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

mat[i][j] = 0;

for (int k = 0; k < len / n; ++k) {

mat[i][j] += (plaintext[k \* n + j] - 'A') \* (ciphertext[k \* n + i] - 'A');

}

mat[i][j] %= 26;

}

}

// Find the determinant of the matrix

int det = (mat[0][0] \* mat[1][1] - mat[0][1] \* mat[1][0] + 26) % 26;

// Find the modular inverse of the determinant

int detInverse = modInverse(det, 26);

if (detInverse == -1) {

printf("Modular inverse does not exist. Unable to recover the key.\n");

return false;

}

// Find the adjugate matrix

int adjMat[MAX\_SIZE][MAX\_SIZE];

adjMat[0][0] = mat[1][1];

adjMat[0][1] = -mat[0][1] + 26;

adjMat[1][0] = -mat[1][0] + 26;

adjMat[1][1] = mat[0][0];

// Find the inverse of the matrix

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

key[i][j] = (adjMat[i][j] \* detInverse) % 26;

}

}

return true;

}

// Function to print a matrix

void printMatrix(const int mat[MAX\_SIZE][MAX\_SIZE], int rows, int cols) {

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

for (int j = 0; j < cols; ++j) {

printf("%d ", mat[i][j]);

}

printf("\n");

}

}

int main() {

const char \*plaintext = "HELLO";

int key[MAX\_SIZE][MAX\_SIZE] = {{6, 24}, {1, 13}};

char ciphertext[MAX\_SIZE];

hillCipherEncrypt(key, plaintext, ciphertext, n);

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

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

printf("\nAttempting Known-Plaintext Attack...\n");

int recoveredKey[MAX\_SIZE][MAX\_SIZE];

if (knownPlaintextAttack(plaintext, ciphertext, recoveredKey, n)) {

printf("Recovered Key:\n");

printMatrix(recoveredKey, n, n);

// Decrypt the ciphertext using the recovered key

char decryptedText[MAX\_SIZE];

hillCipherDecrypt(recoveredKey, ciphertext, decryptedText, n);

printf("\nDecrypted Text: %s\n", decryptedText);

}

return 0;

}

Output:

