











1) #include <stdio.h>

#include <string.h>

int main()

{

char message[100];

int key, i;

printf("Enter a message: ");

gets(message);

printf("Enter the key (shift): ");

scanf("%d", &key);

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

{

char character = message[i];

if (character >= 'a' && character <= 'z')

{

character = (character - 'a' + key) % 26 + 'a';

}

else if (character >= 'A' && character <= 'Z')

{

character = (character - 'A' + key) % 26 + 'A';

}

message[i] = character;

}

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

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

{

char character = message[i];

if (character >= 'a' && character <= 'z')

{

character = (character - 'a' - key + 26) % 26 + 'a';

}

else if (character >= 'A' && character <= 'Z')

{

character = (character - 'A' - key + 26) % 26 + 'A';

}

message[i] = character;

}

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

return 0;

}

2)

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define ALPHABET\_SIZE 26

int main() {

char message[1000];

char key[ALPHABET\_SIZE + 1] = "ZYXWVUTSRQPONMLKJIHGFEDCBA";

printf("Enter the message to encrypt: ");

fgets(message, sizeof(message), stdin);

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

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

if (isalpha(message[i])) {

char base = isupper(message[i]) ? 'A' : 'a';

message[i] = key[message[i] - base];

}

}

printf("Encrypted message: %s", message);

return 0;

}

3)

#include <stdio.h>

#include <string.h>

int main() {

int keyMatrix[3][3];

char input[100];

int len;

printf("Enter the 3x3 key matrix:\n");

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

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

}

}

printf("Enter the plaintext (in uppercase): ");

scanf("%s", input);

len = strlen(input);

while (len % 3 != 0) {

input[len] = 'X';

len++;

}

input[len] = '\0';

int encrypted[len];

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

for (int i = 0; i < len; i += 3) {

for (int j = 0; j < 3; j++) {

encrypted[i + j] = 0;

for (int k = 0; k < 3; k++) {

encrypted[i + j] += keyMatrix[j][k] \* (input[i + k] - 'A');

}

encrypted[i + j] = (encrypted[i + j] % 26 + 26) % 26;

}

}

printf("Ciphertext: ");

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

printf("%c", encrypted[i] + 'A');

}

printf("\n");

return 0;

}

4)

#include<stdio.h>

#include<string.h>

int main(){

char key[100];

char plain[100];

char cipher[100];

printf("Enter key: ");

scanf("%s",key);

printf("Enter plain text: ");

scanf("%s",plain);

int i,l=strlen(plain),l2=strlen(key);

for(i=0;i<l;i++){

char c1=plain[i];

char c2=key[i%l2];

if(c1>='A'&&c1<='Z'){

cipher[i]=((c1-'A')+(c2-'A'))%26+'A';

}

}

printf("Cipher text : %s",cipher);

return 0;

5)

#include <stdio.h>

#include <string.h>

int main()

{

int a = 5;

int b = 4;

char plaintext[] = "HELLO BHARGAV";

int n = strlen(plaintext);

char ciphertext[n + 1];

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

{

char p = plaintext[i];

if(p >= 'A' && p <= 'Z')

{

ciphertext[i] = 'A' + (a \* (p - 'A') + b) % 26;

}

else

{

ciphertext[i] = p;

}

}

ciphertext[n] = '\0';

printf("\nPlaintext = %s\n", plaintext);

printf("\nCiphertext = %s\n", ciphertext);

return 0;

}

6) #include <stdio.h>

int gcd(int a,int b)

{

int temp;

while(b!=0)

{

temp=b;

b=a%b;

a=temp;

}

return a;

}

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;

}

void decrypt\_affine(char ciphertext[],int a,int b)

{

int m=26;

int a\_inv=modInverse(a,m);

if(a\_inv==-1)

{

printf("\nInvalid key\n");

return;

}

int i=0;

while(ciphertext[i])

{

if (ciphertext[i]>='A'&&ciphertext[i]<='Z')

{

char decrypted\_char='A'+(a\_inv\*((ciphertext[i]-'A'-b+m)%m))%m;

printf("%c",decrypted\_char);

}

else

{

printf("%c",ciphertext[i]);

}

i++;

}

printf("\n");

}

int main()

{

char ciphertext[]="XURKX";

int a=7;

int b=10;

decrypt\_affine(ciphertext,a,b);

}

7) #include <stdio.h>

#include <string.h>

#define MAX\_CHARS 256

// Function to count frequency of characters

void countFrequency(const char \*ciphertext, int freq[MAX\_CHARS]) {

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

freq[(unsigned char)ciphertext[i]]++;

}

}

// Function to print frequency of characters

void printFrequency(int freq[MAX\_CHARS]) {

printf("Character Frequencies:\n");

for (int i = 0; i < MAX\_CHARS; i++) {

if (freq[i] > 0) {

printf("'%c': %d\n", i, freq[i]);

}

}

}

// Function to decrypt using a simple substitution cipher

void decrypt(const char \*ciphertext, const char \*substitution, char \*plaintext) {

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

if (ciphertext[i] >= '0' && ciphertext[i] <= '9') {

plaintext[i] = substitution[ciphertext[i] - '0'];

} else {

plaintext[i] = ciphertext[i]; // Keep non-numeric as is

}

}

plaintext[strlen(ciphertext)] = '\0'; // Null-terminate

}

int main() {

const char \*ciphertext = "53‡‡†305))6\*;4826)4‡.)4‡);806\*;48†8¶60))85;;]8\*;:‡\*8†83 (88)5\*†;46(;88\*96\*?;8)\*‡(;485);5\*†2:\*‡(;4956\*2(5\*—4)8¶8\*;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4)485†528806\*81 (‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?";

int freq[MAX\_CHARS] = {0};

countFrequency(ciphertext, freq);

// Print character frequencies

printFrequency(freq);

// Define substitution mapping based on analysis (example only)

// This should be adjusted based on actual analysis

const char substitution[10] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'};

char plaintext[strlen(ciphertext) + 1];

decrypt(ciphertext, substitution, plaintext);

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

return 0;

}

8) #include <stdio.h>

#include <string.h>

#include <ctype.h>

void generateCipher(char \*keyword,char \*cipher)

{

int used[26]={0};

int keyLen=strlen(keyword);

int index=0;

strcpy(cipher,keyword);

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

{

if(isalpha(keyword[i]))

{

keyword[i]=toupper(keyword[i]);

used[keyword[i]-'A']=1;

}

}

for(char ch='A';ch<='Z';ch++)

{

if(!used[ch-'A'])

{

cipher[keyLen+index]=ch;

index++;

}

}

cipher[26]='\0';

}

void encrypt(char \*plaintext,char \*cipher,char \*encrypted)

{

int len=strlen(plaintext);

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

{

if(isalpha(plaintext[i]))

{

char original=toupper(plaintext[i]);

encrypted[i]=cipher[original-'A'];

}

else

{

encrypted[i]=plaintext[i];

}

}

encrypted[len]='\0';

}

int main()

{

char keyword[]="CIPHER";

char plaintext[]="HELLO WORLD";

char cipher[27];

char encrypted[100];

generateCipher(keyword,cipher);

encrypt(plaintext,cipher,encrypted);

printf("\nPlaintext = %s\n",plaintext);

printf("\nEncrypted text = %s\n",encrypted);

}

9)

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define SIZE 200

void findPosition(char keySquare[5][5],char letter,int \*row,int \*col)

{

if(letter=='J')

letter='I';

for(\*row=0;\*row<5;(\*row)++)

{

for(\*col=0;\*col<5;(\*col)++)

{

if(keySquare[\*row][\*col]==letter)

return;

}

}

}

void decryptPlayfair(char \*cipherText,char key[5][5])

{

int len=strlen(cipherText);

char decryptedText[SIZE];

int row1,col1,row2,col2;

for (int i=0;i<len;i+=2)

{

findPosition(key,cipherText[i],&row1,&col1);

findPosition(key,cipherText[i+1],&row2,&col2);

if(row1==row2)

{

decryptedText[i]=key[row1][(col1-1+5)%5];

decryptedText[i+1]=key[row2][(col2-1+5)%5];

}

else if(col1==col2)

{

decryptedText[i]=key[(row1-1+5)%5][col1];

decryptedText[i+1]=key[(row2-1+5)%5][col2];

}

else

{

decryptedText[i]=key[row1][col2];

decryptedText[i+1]=key[row2][col1];

}

}

decryptedText[len]='\0';

printf("\nDecrypted Message = \n%s\n",decryptedText);

}

int main()

{

char key[5][5]=

{

{'K','Y','J','E','X'},

{'U','R','E','B','Z'},

{'W','H','T','Y','F'},

{'S','K','R','O','L'},

{'A','C','D','G','M'}

};

char ciphertext[]="KXJEY UREBE ZWEHE WRYTU HEYFS KREHE GOYFI WTTTU OLKSY CAJPO BOTEI ZONTX BYBNT GONEY CUZWR GDSON SXBOU YWRHE BAAHY USEDQ";

int j=0;

for(int i=0;ciphertext[i];i++)

{

if(ciphertext[i]!=' ')

{

ciphertext[j++]=toupper(ciphertext[i]);

}

}

ciphertext[j]='\0';

decryptPlayfair(ciphertext,key);

}

10) #include <stdio.h>

#include <string.h>

#include <ctype.h>

#define SIZE 100

void preparePlainText(char \*plaintext)

{

int len=strlen(plaintext);

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

{

if(!isalpha(plaintext[i]))

{

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

{

plaintext[j]=plaintext[j+1];

}

len--;

i--;

}

else if(plaintext[i]=='J')

{

plaintext[i]='I';

}

}

if (len%2!=0)

{

plaintext[len++]='X';

}

plaintext[len]='\0';

}

void findPosition(char matrix[5][5],char letter,int \*row,int \*col)

{

if(letter=='J')

{

letter='I';

}

for(\*row=0;\*row<5;(\*row)++)

{

for(\*col=0;\*col<5;(\*col)++)

{

if(matrix[\*row][\*col]==letter)

{

return;

}

}

}

}

void encryptPlayfair(char \*plaintext,char matrix[5][5],char \*encryptedText)

{

int len=strlen(plaintext);

int row1,col1,row2,col2;

int index=0;

for(int i=0;i<len;i+=2)

{

findPosition(matrix,plaintext[i],&row1,&col1);

findPosition(matrix,plaintext[i+1],&row2,&col2);

if(row1==row2)

{

encryptedText[index++]=matrix[row1][(col1+1)%5];

encryptedText[index++]=matrix[row2][(col2+1)%5];

}

else if(col1==col2)

{

encryptedText[index++]=matrix[(row1+1)%5][col1];

encryptedText[index++]=matrix[(row2+1)%5][col2];

}

else

{

encryptedText[index++]=matrix[row1][col2];

encryptedText[index++]=matrix[row2][col1];

}

}

encryptedText[index]='\0';

}

int main()

{

char keyMatrix[5][5] =

{

{'M','F','H','I','K'},

{'U','N','O','P','Q'},

{'Z','V','W','X','Y'},

{'E','L','A','R','G'},

{'D','S','T','B','C'}

};

char plaintext[] ="Must see you over Cadogan West. Coming at once";

char processedText[SIZE];

char encryptedText[SIZE\*2];

strcpy(processedText,plaintext);

preparePlainText(processedText);

encryptPlayfair(processedText,keyMatrix,encryptedText);

printf("\nPlaintext = %s\n",plaintext);

printf("\nEncrypted text = %s\n",encryptedText);

}

11)

#include <stdio.h>

#include <math.h>

int main() {

int n = 25;

unsigned long long fact = 1;

double log2\_fact;

unsigned long long approx\_power\_of\_2;

for (int i = 2; i <= n; i++) {

fact \*= i;

}

log2\_fact = log2(fact);

approx\_power\_of\_2 = round(log2\_fact);

printf("The number of possible keys expressed as an approximate power of 2 is 2^%llu.\n", approx\_power\_of\_2);

return 0;

}  
  
12)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define SIZE 2

void multiplyMatrices(int a[SIZE][SIZE], int b[SIZE][1], int result[SIZE][1]) {

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

result[i][0] = 0;

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

result[i][0] += a[i][k] \* b[k][0];

}

result[i][0] %= 26;

}

}

void inverseMatrix(int key[SIZE][SIZE], int inv[SIZE][SIZE]) {

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

if (det < 0) det += 26;

int invDet = 0;

for (int i = 1; i < 26; i++) {

if ((det \* i) % 26 == 1) {

invDet = i;

break;

}

}

inv[0][0] = (key[1][1] \* invDet) % 26;

inv[0][1] = (-key[0][1] \* invDet) % 26;

inv[1][0] = (-key[1][0] \* invDet) % 26;

inv[1][1] = (key[0][0] \* invDet) % 26;

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

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

if (inv[i][j] < 0) inv[i][j] += 26;

}

}

}

int charToInt(char c) {

return c - 'a';

}

char intToChar(int i) {

return i + 'a';

}

void encryptHillCipher(char\* plaintext, int key[SIZE][SIZE], char\* ciphertext) {

int length = strlen(plaintext);

int plaintextMatrix[SIZE][1];

int keyMatrix[SIZE][SIZE] = {0};

int resultMatrix[SIZE][1];

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

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

keyMatrix[i][j] = key[i][j];

}

}

for (int i = 0; i < length; i += SIZE) {

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

if (i + j < length) {

plaintextMatrix[j][0] = charToInt(plaintext[i + j]);

} else {

plaintextMatrix[j][0] = 0;

}

}

multiplyMatrices(keyMatrix, plaintextMatrix, resultMatrix);

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

ciphertext[i + j] = intToChar(resultMatrix[j][0]);

}

}

ciphertext[length] = '\0';

}

void decryptHillCipher(char\* ciphertext, int key[SIZE][SIZE], char\* plaintext) {

int length = strlen(ciphertext);

int ciphertextMatrix[SIZE][1];

int keyMatrix[SIZE][SIZE] = {0};

int invKeyMatrix[SIZE][SIZE] = {0};

int resultMatrix[SIZE][1];

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

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

keyMatrix[i][j] = key[i][j];

}

}

inverseMatrix(keyMatrix, invKeyMatrix);

for (int i = 0; i < length; i += SIZE) {

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

ciphertextMatrix[j][0] = charToInt(ciphertext[i + j]);

}

multiplyMatrices(invKeyMatrix, ciphertextMatrix, resultMatrix);

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

plaintext[i + j] = intToChar(resultMatrix[j][0]);

}

}

plaintext[length] = '\0';

}

int main() {

char plaintext[100];

int key[SIZE][SIZE] = {{9, 4}, {5, 7}};

char ciphertext[100];

char decryptedtext[100];

printf("Enter plaintext (lowercase letters only): ");

fgets(plaintext, sizeof(plaintext), stdin);

plaintext[strcspn(plaintext, "\n")] = '\0';

char temp[100];

int index = 0;

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

if (plaintext[i] != ' ') {

temp[index++] = plaintext[i];

}

}

temp[index] = '\0';

strcpy(plaintext, temp);

encryptHillCipher(plaintext, key, ciphertext);

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

decryptHillCipher(ciphertext, key, decryptedtext);

printf("Decrypted text: %s\n", decryptedtext);

return 0;

}  
  
13)

#include <stdio.h>

#define MOD 26

int main() {

int plaintext1[2] = {1, 3};

int ciphertext1[2] = {2, 3};

int plaintext2[2] = {1, 2};

int ciphertext2[2] = {1, 4};

int key[2][2];

key[0][0] = (ciphertext1[0] - ciphertext2[0] + MOD) % MOD;

key[0][1] = (ciphertext1[1] - ciphertext2[1] + MOD) % MOD;

key[1][0] = (plaintext1[0] \* key[0][0] + plaintext1[1] \* key[0][1]) % MOD;

key[1][1] = (plaintext2[0] \* key[0][0] + plaintext2[1] \* key[0][1]) % MOD;

if (key[1][0] < 0) key[1][0] += MOD;

if (key[1][1] < 0) key[1][1] += MOD;

printf("Recovered key:\n");

printf("%d %d\n", key[0][0], key[0][1]);

printf("%d %d\n", key[1][0], key[1][1]);

return 0;

}  
  
  
14) #include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

void encryptMessage(char \*plaintext, int \*key, int len) {

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

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

printf("%c", ((plaintext[i] - 'a' + key[i]) % 26) + 'A');

} else if (plaintext[i] >= 'A' && plaintext[i] <= 'Z') {

printf("%c", ((plaintext[i] - 'A' + key[i]) % 26) + 'A');

} else {

printf("%c", plaintext[i]);

}

}

printf("\n");

}

int main() {

char plaintext[100];

int key[100], len;

printf("Enter the plaintext: ");

fgets(plaintext, sizeof(plaintext), stdin);

len = strlen(plaintext);

if (plaintext[len - 1] == '\n') plaintext[--len] = '\0';

srand(time(0));

for (int i = 0; i < len; i++) key[i] = rand() % 26;

printf("Encrypted message: ");

encryptMessage(plaintext, key, len);

return 0;

}  
  
15)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define ALPHABET\_SIZE 26

#define MAX\_LEN 100

#define TOP\_N 10

// Function to convert a character to its numeric representation (0-25)

int charToNum(char c) {

return toupper(c) - 'A';

}

// Function to convert a number (0-25) to its corresponding uppercase character

char numToChar(int num) {

return num + 'A';

}

// Function to perform frequency analysis and return letter frequencies

void analyzeFrequency(const char \*text, int \*frequency) {

int i;

for (i = 0; i < ALPHABET\_SIZE; i++) {

frequency[i] = 0;

}

while (\*text) {

if (isalpha(\*text)) {

frequency[charToNum(\*text)]++;

}

text++;

}

}

// Function to shift ciphertext by a given shift value

void shiftCiphertext(const char \*ciphertext, int shift, char \*plaintext) {

int i;

for (i = 0; ciphertext[i] != '\0'; i++) {

if (isalpha(ciphertext[i])) {

int num = charToNum(ciphertext[i]);

num = (num - shift + ALPHABET\_SIZE) % ALPHABET\_SIZE;

plaintext[i] = numToChar(num);

} else {

plaintext[i] = ciphertext[i];

}

}

plaintext[i] = '\0';

}

// Function to print the top N plaintexts

void printTopPlaintexts(const char \*ciphertext, int topN) {

int i;

char plaintext[MAX\_LEN];

printf("Top %d possible plaintexts:\n", topN);

for (i = 0; i < ALPHABET\_SIZE && i < topN; i++) {

shiftCiphertext(ciphertext, i, plaintext);

printf("%2d: %s\n", i, plaintext);

}

}

int main() {

char ciphertext[MAX\_LEN];

int topN;

// Input ciphertext

printf("Enter the ciphertext (only letters and spaces): ");

fgets(ciphertext, sizeof(ciphertext), stdin);

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

// Input the number of top possible plaintexts to display

printf("Enter the number of top possible plaintexts to display: ");

scanf("%d", &topN);

if (topN <= 0 || topN > ALPHABET\_SIZE) {

printf("Invalid number of top plaintexts.\n");

return 1;

}

// Print the top N possible plaintexts

printTopPlaintexts(ciphertext, topN);

return 0;

}  
  
16)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define ALPHABET\_SIZE 26

#define MAX\_LEN 100

#define TOP\_N 10

// Standard English letter frequency (in percentage)

const double ENGLISH\_FREQ[ALPHABET\_SIZE] = {

8.167, 1.492, 2.782, 4.253, 12.702, 2.228, 2.015, 6.094, 6.966, 0.153,

0.772, 4.025, 2.406, 6.749, 7.507, 1.929, 0.095, 5.987, 6.317, 9.056,

2.758, 0.978, 2.560, 0.150, 1.929, 0.095

};

// Function to convert a character to its numeric representation (0-25)

int charToNum(char c) {

return toupper(c) - 'A';

}

// Function to convert a number (0-25) to its corresponding uppercase character

char numToChar(int num) {

return num + 'A';

}

// Function to calculate letter frequencies in the ciphertext

void calculateFrequencies(const char \*text, int \*freq) {

int i;

for (i = 0; i < ALPHABET\_SIZE; i++) {

freq[i] = 0;

}

while (\*text) {

if (isalpha(\*text)) {

freq[charToNum(\*text)]++;

}

text++;

}

}

// Function to sort frequencies and letters by frequency

void sortFrequencies(int \*freq, char \*letters) {

int i, j;

char temp;

int tempFreq;

for (i = 0; i < ALPHABET\_SIZE - 1; i++) {

for (j = i + 1; j < ALPHABET\_SIZE; j++) {

if (freq[i] < freq[j]) {

// Swap frequencies

tempFreq = freq[i];

freq[i] = freq[j];

freq[j] = tempFreq;

// Swap corresponding letters

temp = letters[i];

letters[i] = letters[j];

letters[j] = temp;

}

}

}

}

// Function to generate a plaintext by applying a substitution

void generatePlaintext(const char \*ciphertext, char \*plaintext, const char \*substitution) {

int i;

for (i = 0; ciphertext[i] != '\0'; i++) {

if (isalpha(ciphertext[i])) {

plaintext[i] = substitution[charToNum(ciphertext[i])];

} else {

plaintext[i] = ciphertext[i];

}

}

plaintext[i] = '\0';

}

// Function to print the top N possible plaintexts

void printTopPlaintexts(const char \*ciphertext, int topN) {

int freq[ALPHABET\_SIZE];

char letters[ALPHABET\_SIZE];

char substitution[ALPHABET\_SIZE];

char plaintext[MAX\_LEN];

int i;

// Initialize letters array

for (i = 0; i < ALPHABET\_SIZE; i++) {

letters[i] = numToChar(i);

}

// Calculate letter frequencies

calculateFrequencies(ciphertext, freq);

// Sort letters by frequency

sortFrequencies(freq, letters);

printf("Top %d possible plaintexts:\n", topN);

for (i = 0; i < topN && i < ALPHABET\_SIZE; i++) {

// Create substitution alphabet

int j;

for (j = 0; j < ALPHABET\_SIZE; j++) {

substitution[j] = letters[(j + i) % ALPHABET\_SIZE]; // Rotate substitution

}

// Generate and print plaintext

generatePlaintext(ciphertext, plaintext, substitution);

printf("%2d: %s\n", i + 1, plaintext);

}

}

int main() {

char ciphertext[MAX\_LEN];

int topN;

// Input ciphertext

printf("Enter the ciphertext (only letters and spaces): ");

fgets(ciphertext, sizeof(ciphertext), stdin);

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

// Input the number of top possible plaintexts to display

printf("Enter the number of top possible plaintexts to display: ");

scanf("%d", &topN);

if (topN <= 0 || topN > TOP\_N) {

printf("Invalid number of top plaintexts.\n");

return 1;

}

// Print the top N possible plaintexts

printTopPlaintexts(ciphertext, topN);

return 0;

}  
  
  
  
17)

#include <stdio.h>

#include <stdint.h>

static const int IP[] = { 2, 6, 3, 1, 4, 8, 5, 7 };

static const int IP\_INV[] = { 4, 1, 3, 5, 7, 2, 8, 6 };

static const uint64\_t KEY = 0x133457799BBCDFF1;

static const uint64\_t CIPHERTEXT = 0x0123456789ABCDEF;

uint64\_t permute(uint64\_t input, const int \*table, int size) {

uint64\_t result = 0;

int i;

for (i = 0; i < size; i++) {

result |= ((input >> (64 - table[i])) & 1) << (size - 1 - i);

}

return result;

}

uint64\_t des\_decrypt(uint64\_t ciphertext, uint64\_t key) {

uint64\_t permuted\_ciphertext = permute(ciphertext, IP, 64);

uint64\_t decrypted = permuted\_ciphertext ^ key;

decrypted = permute(decrypted, IP\_INV, 64);

return decrypted;

}

int main() {

uint64\_t decrypted = des\_decrypt(CIPHERTEXT, KEY);

printf("Ciphertext: 0x%016llX\n", CIPHERTEXT);

printf("Decrypted: 0x%016llX\n", decrypted);

return 0;

}

OUTPUT:

Ciphertext: 0x0123456789ABCDEF

Decrypted: 0x8B231170CDBC0B80  
  
18) #include <stdio.h>

#include <stdint.h>

uint64\_t key = 0x133457799BBCDFF1;

uint64\_t plaintext = 0x0123456789ABCDEF;

int initial\_permutation[] = {

58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6,

64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1,

59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7

};

uint32\_t des\_round(uint32\_t data, uint32\_t round\_key)

{

uint32\_t expanded\_data = 0;

uint32\_t substituted\_data = expanded\_data ^ round\_key;

return substituted\_data;

}

int main()

{

int i;

uint64\_t permuted\_data = 0;

for (i = 0; i < 64; i++)

{

int bit\_pos = initial\_permutation[i] - 1;

uint64\_t bit = (plaintext >> bit\_pos) & 1;

permuted\_data |= (bit << (63 - i));

}

uint32\_t left = permuted\_data >> 32;

uint32\_t right = (uint32\_t) permuted\_data;

uint32\_t round\_key = (uint32\_t) (key >> 32);

uint32\_t new\_right = des\_round(right, round\_key);

uint64\_t ciphertext = ((uint64\_t) left << 32) | new\_right;

printf("Plaintext: 0x%016llX\n", plaintext);

printf("Ciphertext: 0x%016llX\n", ciphertext);

return 0;

}

output:

Plaintext: 0x0123456789ABCDEF

Ciphertext: 0x550F550F13345779

19)

#include <stdio.h>

#include <string.h>

typedef unsigned char byte;

// Function to perform 3DES encryption in CBC mode

void encrypt3DES\_CBC(const byte \*plaintext, const byte \*key, const byte \*iv, byte \*ciphertext) {

// Encrypt the plaintext using 3DES in CBC mode

// You need to implement the 3DES algorithm here

// For simplicity, let's just copy plaintext to ciphertext (replace this with actual 3DES encryption)

memcpy(ciphertext, plaintext, 8);

}

int main() {

const byte plaintext[] = "Hello123"; // Replace with your plaintext

const byte key[] = {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF}; // 3DES key (24 bytes)

const byte iv[] = {0x12, 0x34, 0x56, 0x78, 0x90, 0xAB, 0xCD, 0xEF}; // Initialization Vector (IV) (8 bytes)

byte ciphertext[64]; // Output buffer

// Encrypt in CBC mode using 3DES

encrypt3DES\_CBC(plaintext, key, iv, ciphertext);

// Display the ciphertext

printf("Ciphertext: ");

for (int i = 0; i < 64; ++i) {

printf("%02X", (unsigned char)ciphertext[i]);

}

printf("\n");

return 0;

}

OUTPUT:

Ciphertext: 48656C6C6F31323300000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000005D00C1100000000  
  
  
20)

#include <stdio.h>

#include <string.h>

// Function to simulate CBC encryption with error propagation

void cbcEncryptWithErrorPropagation(const char \*plaintext, char \*ciphertext, size\_t blockCount) {

char iv[16] = {0}; // Initialization vector

for (size\_t i = 0; i < blockCount; ++i) {

// Simulate XOR with previous ciphertext block

for (int j = 0; j < 16; ++j) {

ciphertext[i \* 16 + j] = plaintext[i \* 16 + j] ^ iv[j];

}

// Simulate encryption (replace this with actual encryption logic)

// In a real-world scenario, you would use a proper encryption function.

// Here, we just copy the XORed value as a placeholder.

memcpy(iv, ciphertext + i \* 16, 16);

}

}

int main() {

const char plaintext[] = "This is a sample plaintext block with an error in the first block.";

const size\_t blockCount = strlen(plaintext) / 16 + 1; // Calculate the number of blocks needed

char ciphertext[256]; // Output buffer

// Encrypt in CBC mode with error propagation

cbcEncryptWithErrorPropagation(plaintext, ciphertext, blockCount);

// Display the ciphertext

printf("Ciphertext:\n");

for (size\_t i = 0; i < blockCount; ++i) {

for (int j = 0; j < 16; ++j) {

printf("%02X", (unsigned char)ciphertext[i \* 16 + j]);

}

printf(" ");

}

printf("\n");

return 0;

}

OUTPUT:

Ciphertext:

5468697320697320612073616D706C65 741805124907074519545303011F0F0E 546F6C662127662B393121716E6D2F67 3A4F180E440700424B4255510C014004 5161180E440700424E4255510C014004  
  
  
  
21)

#include <stdio.h>

#include <string.h>

// Function prototypes

void encrypt\_ecb(char \*plaintext, char \*key, int length);

void decrypt\_ecb(char \*ciphertext, char \*key, int length);

int main() {

char plaintext[] = "HELLOWOR";

char key[] = "KEY12345";

int length = strlen(plaintext);

// ECB Mode

encrypt\_ecb(plaintext, key, length);

decrypt\_ecb(plaintext, key, length);

return 0;

}

// ECB Mode Encryption

void encrypt\_ecb(char \*plaintext, char \*key, int length) {

printf("\nECB Encryption:\n");

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

plaintext[i] = plaintext[i] ^ key[i % strlen(key)];

}

printf("Ciphertext: ");

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

printf("%02x", plaintext[i]);

}

printf("\n");

}

// ECB Mode Decryption

void decrypt\_ecb(char \*ciphertext, char \*key, int length) {

printf("\nECB Decryption:\n");

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

ciphertext[i] = ciphertext[i] ^ key[i % strlen(key)];

}

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

}

OUTPUT:

ECB Encryption:

Ciphertext: 0300157d7d647b67

ECB Decryption:

Decrypted Text: HELLOWOR  
  
22)

#include <stdio.h>

// Function prototypes

void encrypt\_sdes\_cbc(char \*plaintext, char \*key, char \*iv, int length);

void decrypt\_sdes\_cbc(char \*ciphertext, char \*key, char \*iv, int length);

void sdes\_encrypt(char \*block, char \*key);

void sdes\_decrypt(char \*block, char \*key);

int main() {

char plaintext[] = "0000000100100011"; // Binary plaintext

char key[] = "0111111101"; // Binary key

char iv[] = "10101010"; // Binary initialization vector

int length = 16; // Manually set the length

// S-DES CBC Mode Encryption

encrypt\_sdes\_cbc(plaintext, key, iv, length);

// S-DES CBC Mode Decryption

decrypt\_sdes\_cbc(plaintext, key, iv, length);

return 0;

}

// S-DES Encryption

void sdes\_encrypt(char \*block, char \*key) {

// Implement S-DES encryption logic here

// This function should modify the 'block' in-place

// based on the provided 'key'

// Replace the following comment with your S-DES encryption implementation

// Example (not actual S-DES): block = block XOR key;

}

// S-DES Decryption

void sdes\_decrypt(char \*block, char \*key) {

// Implement S-DES decryption logic here

// This function should modify the 'block' in-place

// based on the provided 'key'

// Replace the following comment with your S-DES decryption implementation

// Example (not actual S-DES): block = block XOR key;

}

// CBC Mode Encryption

void encrypt\_sdes\_cbc(char \*plaintext, char \*key, char \*iv, int length) {

printf("\nS-DES CBC Encryption:\n");

// Convert IV and plaintext to integers or use bitwise XOR directly

// Perform encryption block by block using CBC mode

for (int i = 0; i < length; i += 8) {

for (int j = 0; j < 8; j++) {

plaintext[i + j] = plaintext[i + j] ^ iv[j]; // XOR with IV

}

sdes\_encrypt(plaintext + i, key);

// IV for the next block becomes the ciphertext of the current block

for (int j = 0; j < 8; j++) {

iv[j] = plaintext[i + j];

}

}

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

}

// CBC Mode Decryption

void decrypt\_sdes\_cbc(char \*ciphertext, char \*key, char \*iv, int length) {

printf("\nS-DES CBC Decryption:\n");

char prev\_ciphertext[9]; // Store the ciphertext of the previous block

// Convert IV and ciphertext to integers or use bitwise XOR directly

// Perform decryption block by block using CBC mode

for (int i = 0; i < length; i += 8) {

for (int j = 0; j < 8; j++) {

prev\_ciphertext[j] = ciphertext[i + j]; // Store current ciphertext

}

sdes\_decrypt(ciphertext + i, key);

for (int j = 0; j < 8; j++) {

ciphertext[i + j] = ciphertext[i + j] ^ iv[j]; // XOR with IV

}

// IV for the next block becomes the ciphertext of the current block

for (int j = 0; j < 8; j++) {

iv[j] = prev\_ciphertext[j];

}

}

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

}

OUTPUT:

S-DES CBC Encryption:

Ciphertext:

S-DES CBC Decryption:

Decrypted Text: 0010001100100011

23)

#include <stdio.h>

// Function prototypes

void encrypt\_sdes\_cbc(char \*plaintext, char \*key, char \*iv, int length);

void decrypt\_sdes\_cbc(char \*ciphertext, char \*key, char \*iv, int length);

void sdes\_encrypt(char \*block, char \*key);

void sdes\_decrypt(char \*block, char \*key);

int main() {

char plaintext[] = "0000000100100011"; // Binary plaintext

char key[] = "0111111101"; // Binary key

char iv[] = "10101010"; // Binary initialization vector

int length = 16; // Manually set the length

// S-DES CBC Mode Encryption

encrypt\_sdes\_cbc(plaintext, key, iv, length);

// S-DES CBC Mode Decryption

decrypt\_sdes\_cbc(plaintext, key, iv, length);

return 0;

}

// S-DES Encryption

void sdes\_encrypt(char \*block, char \*key) {

// Implement S-DES encryption logic here

// This function should modify the 'block' in-place

// based on the provided 'key'

// Replace the following comment with your S-DES encryption implementation

// Example (not actual S-DES): block = block XOR key;

}

// S-DES Decryption

void sdes\_decrypt(char \*block, char \*key) {

// Implement S-DES decryption logic here

// This function should modify the 'block' in-place

// based on the provided 'key'

// Replace the following comment with your S-DES decryption implementation

// Example (not actual S-DES): block = block XOR key;

}

// CBC Mode Encryption

void encrypt\_sdes\_cbc(char \*plaintext, char \*key, char \*iv, int length) {

printf("\nS-DES CBC Encryption:\n");

// Convert IV and plaintext to integers or use bitwise XOR directly

// Perform encryption block by block using CBC mode

for (int i = 0; i < length; i += 8) {

for (int j = 0; j < 8; j++) {

plaintext[i + j] = plaintext[i + j] ^ iv[j]; // XOR with IV

}

sdes\_encrypt(plaintext + i, key);

// IV for the next block becomes the ciphertext of the current block

for (int j = 0; j < 8; j++) {

iv[j] = plaintext[i + j];

}

}

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

}

// CBC Mode Decryption

void decrypt\_sdes\_cbc(char \*ciphertext, char \*key, char \*iv, int length) {

printf("\nS-DES CBC Decryption:\n");

char prev\_ciphertext[9]; // Store the ciphertext of the previous block

// Convert IV and ciphertext to integers or use bitwise XOR directly

// Perform decryption block by block using CBC mode

for (int i = 0; i < length; i += 8) {

for (int j = 0; j < 8; j++) {

prev\_ciphertext[j] = ciphertext[i + j]; // Store current ciphertext

}

sdes\_decrypt(ciphertext + i, key);

for (int j = 0; j < 8; j++) {

ciphertext[i + j] = ciphertext[i + j] ^ iv[j]; // XOR with IV

}

// IV for the next block becomes the ciphertext of the current block

for (int j = 0; j < 8; j++) {

iv[j] = prev\_ciphertext[j];

}

}

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

}

OUTPUT:

S-DES CBC Encryption:

Ciphertext:

S-DES CBC Decryption:

Decrypted Text: 0010001100100011  
  
  
24)

#include <stdio.h>

// Function to find the greatest common divisor (GCD)

int gcd(int a, int b) {

if (b == 0)

return a;

return gcd(b, a % b);

}

// Function to find the multiplicative inverse using the extended Euclidean algorithm

int modInverse(int a, int m) {

int m0 = m, t, q;

int x0 = 0, x1 = 1;

if (m == 1)

return 0;

while (a > 1) {

q = a / m;

t = m;

m = a % m;

a = t;

t = x0;

x0 = x1 - q \* x0;

x1 = t;

}

if (x1 < 0)

x1 += m0;

return x1;

}

int main() {

int e = 31;

int n = 3599;

// Trial-and-error to find p and q (you might want a more sophisticated method in practice)

int p = 61;

int q = 59;

int phi\_n = (p - 1) \* (q - 1);

// Check if e and phi\_n are coprime

if (gcd(e, phi\_n) != 1) {

printf("Invalid choice of e. Choose a different value.\n");

return 1;

}

// Calculate the private key d using the extended Euclidean algorithm

int d = modInverse(e, phi\_n);

printf("Public key: (e=%d, n=%d)\n", e, n);

printf("Private key: (d=%d, n=%d)\n", d, n);

return 0;

}

OUTPUT:

Public key: (e=31, n=3599)

Private key: (d=3031, n=3599)  
  
25)

#include <stdio.h>

// Function to calculate the greatest common divisor (GCD) using Euclid's algorithm

int gcd(int a, int b) {

if (b == 0)

return a;

return gcd(b, a % b);

}

// Function to factorize n given a non-trivial factor

void factorize(int n, int factor) {

int p = factor;

int q = n / factor;

printf("Factorization of n: p = %d, q = %d\n", p, q);

}

int main() {

// Assume n = pq, where p and q are large prime numbers

int p = 60;

int q = 53;

int n = p \* q;

// Assume e is the public key

int e = 17;

// Assume we have a ciphertext block that has a common factor with n

int ciphertext = 123;

// Check if the gcd(ciphertext, n) is not 1

if (gcd(ciphertext, n) != 1) {

printf("The ciphertext has a common factor with n.\n");

// Factorize n using the non-trivial factor

factorize(n, gcd(ciphertext, n));

} else {

printf("The ciphertext does not have a common factor with n.\n");

}

return 0;

}

OUTPUT:

The ciphertext has a common factor with n.

Factorization of n: p = 3, q = 1060  
  
  
26)

#include <stdio.h>

// Function to calculate the greatest common divisor (GCD) using Euclid's algorithm

int gcd(int a, int b) {

if (b == 0)

return a;

return gcd(b, a % b);

}

// Function to calculate the multiplicative inverse using the extended Euclidean algorithm

int modInverse(int a, int m) {

int m0 = m, t, q;

int x0 = 0, x1 = 1;

if (m == 1)

return 0;

while (a > 1) {

q = a / m;

t = m;

m = a % m;

a = t;

t = x0;

x0 = x1 - q \* x0;

x1 = t;

}

if (x1 < 0)

x1 += m0;

return x1;

}

// Function to generate a new public key (e, n) and private key (d, n)

void generateNewKeys(int \*e, int \*d, int \*n) {

// Choose new prime numbers p and q (in practice, use a secure random number generator)

int p = 60;

int q = 53;

// Calculate n and phi(n)

\*n = p \* q;

int phi\_n = (p - 1) \* (q - 1);

// Choose a new public exponent e (in practice, choose a suitable value)

\*e = 17;

// Calculate the private key d

\*d = modInverse(\*e, phi\_n);

}

int main() {

int original\_e = 17;

int original\_d = 2853; // Example leaked private key

int original\_n = 3233; // Example modulus

// Output original public and private keys

printf("Original Public Key: (e=%d, n=%d)\n", original\_e, original\_n);

printf("Original Private Key: (d=%d, n=%d)\n", original\_d, original\_n);

// Generate new public and private keys after a private key leak

int new\_e, new\_d, new\_n;

generateNewKeys(&new\_e, &new\_d, &new\_n);

// Output new public and private keys

printf("\nNew Public Key: (e=%d, n=%d)\n", new\_e, new\_n);

printf("New Private Key: (d=%d, n=%d)\n", new\_d, new\_n);

return 0;

}

OUTPUT:

Original Public Key: (e=17, n=3233)

Original Private Key: (d=2853, n=3233)

New Public Key: (e=17, n=3180)

New Private Key: (d=361, n=3180)  
  
  
27)

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// Function to calculate the greatest common divisor (GCD) using Euclid's algorithm

int gcd(int a, int b) {

if (b == 0)

return a;

return gcd(b, a % b);

}

// Function to calculate the multiplicative inverse using the extended Euclidean algorithm

int modInverse(int a, int m) {

int m0 = m, t, q;

int x0 = 0, x1 = 1;

if (m == 1)

return 0;

while (a > 1) {

q = a / m;

t = m;

m = a % m;

a = t;

t = x0;

x0 = x1 - q \* x0;

x1 = t;

}

if (x1 < 0)

x1 += m0;

return x1;

}

// Function to calculate modular exponentiation (to handle large numbers)

int modExp(int base, int exponent, int modulus) {

int result = 1;

base = base % modulus;

while (exponent > 0) {

if (exponent % 2 == 1)

result = (result \* base) % modulus;

exponent = exponent >> 1;

base = (base \* base) % modulus;

}

return result;

}

int main() {

// Assume Bob's public key (e, n)

int e = 65537; // Commonly used value for e

int n = 309485009; // A large modulus (product of two large primes)

// Message to be encrypted (each character represented as an integer between 0 and 25)

int plaintext[] = {3, 2, 11, 18, 4};

// Encrypt each character separately

int ciphertext[5];

for (int i = 0; i < 5; ++i) {

ciphertext[i] = modExp(plaintext[i], e, n);

}

// Display the encrypted message

printf("Encrypted Message: ");

for (int i = 0; i < 5; ++i) {

printf("%d ", ciphertext[i]);

}

printf("\n");

return 0;

}

OUTPUT:

Encrypted Message: -33500114 0 -195051537 -34723648 0  
  
28)

#include<stdio.h>

#include<cmath>

int main(){

int q=11,g=2;

int xa,xb;

int ya,yb;

int ka,kb;

printf("enter sender private key \n");

scanf("%d",&xa);

printf("enter receiver private key \n");

scanf("%d",&xb);

int a = pow(g,xa);

ya = a % q ;

int b = pow(g,xb);

yb = b % q;

printf("public key of sender is %d",ya);

printf("public key of reciver is %d",yb);

int c = pow(ya,xb);

int d = pow(yb,xa);

ka = c % q;

kb = d % q;

if(ka == kb){

printf("passes");

}else{

printf("failed");

}

return 0;

}

29)

#include <stdio.h>

#include <stdint.h>

#define STATE\_WIDTH 5

#define STATE\_HEIGHT 5

#define LANE\_SIZE 64

#define STATE\_SIZE (STATE\_WIDTH \* STATE\_HEIGHT)

#define CAPACITY\_LANES 1024

#define ROUND\_COUNT 24

typedef struct {

uint64\_t state[STATE\_SIZE];

} SHA3State;

// Function prototypes

void keccakPermutation(SHA3State \*state);

void theta(SHA3State \*state);

void rho(SHA3State \*state);

void pi(SHA3State \*state);

void chi(SHA3State \*state);

void iota(SHA3State \*state, int round);

int main() {

SHA3State sha3State;

// Initialize state (for simplicity, set a lane to a non-zero value)

sha3State.state[0] = 0x1;

// Perform the SHA-3 permutation

for (int round = 0; round < ROUND\_COUNT; round++) {

keccakPermutation(&sha3State);

}

// Output the state

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

printf("%016lx ", sha3State.state[i]);

if ((i + 1) % STATE\_WIDTH == 0) {

printf("\n");

}

}

return 0;

}

// Placeholder for the actual SHA-3 permutation (Keccak-f[1600])

void keccakPermutation(SHA3State \*state) {

theta(state);

rho(state);

pi(state);

chi(state);

iota(state, 0);

}

// Placeholder function for the theta step

void theta(SHA3State \*state) {

// Simple XOR operation as a placeholder

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

state->state[i] ^= 0x123456789abcdef0;

}

}

// Placeholder function for the rho step

void rho(SHA3State \*state) {

// Simple shift as a placeholder

state->state[0] <<= 1;

}

// Placeholder function for the pi step

void pi(SHA3State \*state) {

// Simple permutation as a placeholder

uint64\_t temp = state->state[0];

state->state[0] = state->state[1];

state->state[1] = temp;

}

// Placeholder function for the chi step

void chi(SHA3State \*state) {

// Simple AND operation as a placeholder

state->state[0] &= 0x0f0f0f0f0f0f0f0f;

}

// Placeholder function for the iota step

void iota(SHA3State \*state, int round) {

// XOR with a round constant as a placeholder

state->state[0] ^= round;

}

OUTPUT:

0204060705030100 2060a0ff3f7fbfe0 0000034000000340 0000034000000340 0000034000000340

0000034000000340 0000000000000000 0000000000000100 0000000000000000 0000000000000000

0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000

0000000000000000 0000000000000000 00000000110cd005 0000000000400040 0000000000401365

0000000000000000 0000000000000000 0000000000401320 0000000000401050 00007ffc67e24e20

30)

#include <stdio.h>

#include <string.h>

#include <stdlib.h> // Include this header for malloc

#define BLOCK\_SIZE 16

void xorBlocks(unsigned char \*result, const unsigned char \*block1, const unsigned char \*block2) {

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

result[i] = block1[i] ^ block2[i];

}

}

void aesEncrypt(const unsigned char \*input, const unsigned char \*key, unsigned char \*output) {

// TODO: Implement your own AES encryption logic here

// This is a placeholder and needs to be replaced with a secure AES implementation

}

void cbcMac(const unsigned char \*message, const unsigned char \*key, unsigned char \*mac, size\_t messageSize) {

unsigned char iv[BLOCK\_SIZE] = {0}; // Initialization vector for CBC

unsigned char ciphertext[BLOCK\_SIZE];

// Encrypt the one-block message using AES in CBC mode

aesEncrypt(message, key, ciphertext);

xorBlocks(mac, message, ciphertext);

// Encrypt the two-block message X || (X ? T) using AES in CBC mode

unsigned char twoBlockMessage[2 \* BLOCK\_SIZE];

memcpy(twoBlockMessage, message, BLOCK\_SIZE);

xorBlocks(twoBlockMessage + BLOCK\_SIZE, message, mac);

aesEncrypt(twoBlockMessage, key, mac);

}

int main() {

unsigned char key[BLOCK\_SIZE] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x97, 0x7b, 0xab, 0xf7, 0x97, 0x7b};

// Read the message from a file

FILE \*file = fopen("message.txt", "rb");

if (file == NULL) {

perror("Error opening file");

return 1;

}

fseek(file, 0, SEEK\_END);

size\_t fileSize = ftell(file);

fseek(file, 0, SEEK\_SET);

unsigned char \*message = (unsigned char \*)malloc(fileSize); // Include a cast

if (message == NULL) {

perror("Error allocating memory");

fclose(file);

return 1;

}

fread(message, 1, fileSize, file);

fclose(file);

unsigned char mac[BLOCK\_SIZE];

// Calculate CBC-MAC for the message

cbcMac(message, key, mac, fileSize);

// Print the resulting MAC

printf("CBC-MAC for the message: ");

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

printf("%02x", mac[i]);

}

printf("\n");

free(message); // Don't forget to free allocated memory

return 0;

}  
  
  
31)

#include <stdio.h>

#define BLOCK\_SIZE 64

// Function to perform left shift on a block

void leftShift(unsigned char \*block, int size) {

int i, carry = 0;

for (i = 0; i < size; i++) {

int nextCarry = (block[i] & 0x80) ? 1 : 0; // Check if the leftmost bit is 1

block[i] = (block[i] << 1) | carry;

carry = nextCarry;

}

}

// Function to perform XOR with a constant

void xorWithConstant(unsigned char \*block, int size, unsigned char constant) {

block[size - 1] ^= constant;

}

// Function to generate CMAC subkeys

void generateCMACSubkeys(unsigned char \*subkey1, unsigned char \*subkey2) {

// Step 1: Apply the block cipher to a block of 0 bits

unsigned char zeroBlock[BLOCK\_SIZE / 8] = {0}; // Initialize a block of 0 bits

// Assuming you have a function named 'blockCipher' to apply the block cipher

// blockCipher(zeroBlock, key, encryptedBlock);

// Step 2: Left shift and XOR with constant for subkey1

leftShift(zeroBlock, BLOCK\_SIZE / 8);

xorWithConstant(zeroBlock, BLOCK\_SIZE / 8, 0x87); // Constant for a block size of 64 bits

// Copy the result to subkey1

for (int i = 0; i < BLOCK\_SIZE / 8; i++) {

subkey1[i] = zeroBlock[i];

}

// Step 3: Left shift and XOR with constant for subkey2

leftShift(subkey1, BLOCK\_SIZE / 8);

if (subkey1[0] & 0x80) {

xorWithConstant(subkey1, BLOCK\_SIZE / 8, 0x1B); // Constant for a block size of 64 bits

}

// Copy the result to subkey2

for (int i = 0; i < BLOCK\_SIZE / 8; i++) {

subkey2[i] = subkey1[i];

}

}

int main() {

unsigned char subkey1[BLOCK\_SIZE / 8];

unsigned char subkey2[BLOCK\_SIZE / 8];

generateCMACSubkeys(subkey1, subkey2);

// Print the generated subkeys

printf("Subkey 1: ");

for (int i = 1; i < BLOCK\_SIZE / 8; i++) {

printf("%02x ", subkey1[i]);

}

printf("\n");

printf("Subkey 2: ");

for (int i = 1; i < BLOCK\_SIZE / 8; i++) {

printf("%02x ", subkey2[i]);

}

printf("\n");

return 0;

}  
  
32)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Simple DSA-like structure for educational purposes

typedef struct {

unsigned long long p;

unsigned long long q;

unsigned long long g;

unsigned long long x; // private key

unsigned long long y; // public key

} DSAKey;

typedef struct {

unsigned long long r;

unsigned long long s;

} DSASignature;

// Simple RSA-like structure for educational purposes

typedef struct {

unsigned long long n;

unsigned long long e; // public exponent

unsigned long long d; // private exponent

} RSAKey;

// Simple structure to represent a message

typedef struct {

unsigned char \*data;

size\_t length;

} Message;

unsigned long long mod\_exp(unsigned long long base, unsigned long long exponent, unsigned long long modulus) {

unsigned long long result = 1;

base = base % modulus;

while (exponent > 0) {

if (exponent % 2 == 1) {

result = (result \* base) % modulus;

}

exponent = exponent >> 1;

base = (base \* base) % modulus;

}

return result;

}

void generate\_dsa\_key(DSAKey \*key) {

// TODO: Implement DSA key generation

}

void generate\_rsa\_key(RSAKey \*key) {

// TODO: Implement RSA key generation

}

DSASignature \*generate\_dsa\_signature(const Message \*message, const DSAKey \*key) {

DSASignature \*signature = (DSASignature \*)malloc(sizeof(DSASignature));

// TODO: Implement DSA signature generation

return signature;

}

unsigned long long generate\_rsa\_signature(const Message \*message, const RSAKey \*key) {

// TODO: Implement RSA signature generation

return 0;

}

int main() {

// Message to be signed

Message message;

message.data = (unsigned char \*)"Hello, DSA and RSA!";

message.length = strlen((char \*)message.data);

// Generate DSA key pair

DSAKey dsaKey;

generate\_dsa\_key(&dsaKey);

// Generate DSA signature for the message

DSASignature \*dsaSignature = generate\_dsa\_signature(&message, &dsaKey);

// Print DSA signature

printf("DSA Signature:\n");

printf("r: %llu\n", dsaSignature->r);

printf("s: %llu\n", dsaSignature->s);

// Generate RSA key pair

RSAKey rsaKey;

generate\_rsa\_key(&rsaKey);

// Generate RSA signature for the same message

unsigned long long rsaSignature = generate\_rsa\_signature(&message, &rsaKey);

// Print RSA signature

printf("RSA Signature: %llu\n", rsaSignature);

// Clean up

free(dsaSignature);

return 0;

}

OUTOUT:

DSA Signature:

r: 0

s: 0

RSA Signature: 0  
  
33)

#include <stdio.h>

#include <stdint.h>

// Initial permutation table (IP)

const uint8\_t initial\_permutation\_table[] = {

58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6,

64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1,

59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7

};

// Initial permutation inverse table (IP^-1)

const uint8\_t initial\_permutation\_inverse\_table[] = {

40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30,

37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28,

35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26,

33, 1, 41, 9, 49, 17, 57, 25

};

// Perform the initial permutation

void initialPermutation(uint64\_t \*block) {

uint64\_t result = 0;

for (int i = 0; i < 64; i++) {

result |= ((\*block >> (64 - initial\_permutation\_table[i])) & 1) << (63 - i);

}

\*block = result;

}

// Perform the initial permutation inverse

void initialPermutationInverse(uint64\_t \*block) {

uint64\_t result = 0;

for (int i = 0; i < 64; i++) {

result |= ((\*block >> (64 - initial\_permutation\_inverse\_table[i])) & 1) << (63 - i);

}

\*block = result;

}

int main() {

uint64\_t plaintext = 0x0123456789ABCDEF; // 64-bit plaintext

printf("Original Plaintext: %016llX\n", plaintext);

// Initial permutation

initialPermutation(&plaintext);

printf("After Initial Permutation: %016llX\n", plaintext);

// Initial permutation inverse

initialPermutationInverse(&plaintext);

printf("After Initial Permutation Inverse: %016llX\n", plaintext);

return 0;

}

OUTPUT:

Original Plaintext: 0123456789ABCDEF

After Initial Permutation: CC00CCFFF0AAF0AA

After Initial Permutation Inverse: 0123456789ABCDEF  
  
  
34)

#include <stdio.h>

#include <stdint.h>

// AES constants

#define AES\_BLOCK\_SIZE 16

// AES S-box

static const uint8\_t sbox[256] = {

// ... (omitted for brevity)

};

// AES encryption round keys

static const uint8\_t roundKeys[11][4][4] = {

// ... (omitted for brevity)

};

// AES encryption function

void aes\_encrypt\_block(const uint8\_t \*input, const uint8\_t \*key, uint8\_t \*output) {

// Initial round key addition

// ... (omitted for brevity)

// Main rounds

// ... (omitted for brevity)

// Final round (no MixColumns)

// ... (omitted for brevity)

}

// ECB mode encryption

void ecb\_encrypt(const uint8\_t \*plaintext, size\_t size, const uint8\_t \*key, uint8\_t \*ciphertext) {

for (size\_t i = 0; i < size; i += AES\_BLOCK\_SIZE) {

aes\_encrypt\_block(plaintext + i, key, ciphertext + i);

}

}

int main() {

const uint8\_t plaintext[] = "This is a sample message";

const uint8\_t key[] = "0123456789abcdef";

uint8\_t ciphertext[128];

// ECB Mode Encryption

ecb\_encrypt(plaintext, sizeof(plaintext) - 1, key, ciphertext);

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

printf("Ciphertext: ");

for (size\_t i = 0; i < sizeof(plaintext) - 1; ++i) {

printf("%02x", ciphertext[i]);

}

printf("\n");

return 0;

}

OUTPUT:

Plaintext: This is a sample message

Ciphertext: 400300004003000040030000400300004003000040030000  
  
  
35)

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <time.h>

#define ALPHABET\_SIZE 26

// Function to generate a random key of given length

void generateRandomKey(int \*key, int length) {

srand((unsigned int)time(NULL));

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

key[i] = rand() % ALPHABET\_SIZE;

}

}

// Function to encrypt plaintext using the one-time pad Vigenère cipher

void encryptVigenere(const char \*plaintext, int \*key, char \*ciphertext) {

int keyIndex = 0;

for (size\_t i = 0; plaintext[i] != '\0'; ++i) {

if (isalpha(plaintext[i])) {

char base = isupper(plaintext[i]) ? 'A' : 'a';

ciphertext[i] = (plaintext[i] - base + key[keyIndex]) % ALPHABET\_SIZE + base;

keyIndex = (keyIndex + 1) % ALPHABET\_SIZE;

} else {

// Non-alphabetic characters remain unchanged

ciphertext[i] = plaintext[i];

}

}

}

int main() {

const char \*plaintext = "Hello, World!";

int keyLength = 12; // Choose an appropriate key length

int key[keyLength];

char ciphertext[256];

// Generate a random key

generateRandomKey(key, keyLength);

// Encrypt the plaintext

encryptVigenere(plaintext, key, ciphertext);

// Display the results

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

printf("Key: ");

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

printf("%d ", key[i]);

}

printf("\n");

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

return 0;

}

OUTPUT:

Plaintext: Hello, World!

Key: 9 25 1 3 20 17 11 17 8 4 9 17

Ciphertext: Qdmoi, Nzith!

36)

#include <stdio.h>

#include <ctype.h>

// Function to calculate the modular multiplicative inverse

int modInverse(int a, int m) {

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

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

return x;

}

}

return -1; // Inverse does not exist

}

// Function to encrypt a single character using the affine Caesar cipher

char encryptChar(int a, int b, char plaintext) {

if (isalpha(plaintext)) {

char base = isupper(plaintext) ? 'A' : 'a';

return (a \* (plaintext - base) + b) % 26 + base;

}

return plaintext; // Non-alphabetic characters remain unchanged

}

// Function to encrypt a string using the affine Caesar cipher

void encryptAffineCaesar(int a, int b, const char \*plaintext, char \*ciphertext) {

for (size\_t i = 0; plaintext[i] != '\0'; ++i) {

ciphertext[i] = encryptChar(a, b, plaintext[i]);

}

}

int main() {

int a = 5; // Choose an appropriate value for a (must be coprime with 26)

int b = 8; // Choose any value for b

const char \*plaintext = "Hello, World!";

char ciphertext[256];

// Encrypt the plaintext

encryptAffineCaesar(a, b, plaintext, ciphertext);

// Display the results

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

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

return 0;

}

OUTPUT:

Plaintext: Hello, World!

Ciphertext: Rclla, Oaplx!  
  
37)

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define ALPHABET\_SIZE 26

// English letter frequency distribution (in percent)

const float englishFreq[ALPHABET\_SIZE] = {

8.167, 1.492, 2.782, 4.253, 12.702, 2.228, 2.015, 6.094,

6.966, 0.153, 0.772, 4.025, 2.406, 6.749, 7.507, 1.929,

0.095, 5.987, 6.327, 9.056, 2.758, 0.978, 2.360, 0.150,

1.974, 0.074

};

// Function to calculate the chi-squared statistic for a given distribution

float calculateChiSquared(const int \*observedFreq) {

float chiSquared = 0.0;

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

chiSquared += ((observedFreq[i] - englishFreq[i]) \* (observedFreq[i] - englishFreq[i])) / englishFreq[i];

}

return chiSquared;

}

// Function to decrypt a monoalphabetic substitution cipher with a given key

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

for (size\_t i = 0; ciphertext[i] != '\0'; ++i) {

if (isalpha(ciphertext[i])) {

char base = isupper(ciphertext[i]) ? 'A' : 'a';

plaintext[i] = (ciphertext[i] - key[ciphertext[i] - base] + 26) % 26 + base;

} else {

plaintext[i] = ciphertext[i];

}

}

}

// Function to perform a letter frequency attack on a monoalphabetic substitution cipher

void letterFrequencyAttack(const char \*ciphertext, int topPlaintexts) {

int observedFreq[ALPHABET\_SIZE] = {0};

int totalLetters = 0;

// Calculate observed letter frequencies in the ciphertext

for (size\_t i = 0; ciphertext[i] != '\0'; ++i) {

if (isalpha(ciphertext[i])) {

char base = isupper(ciphertext[i]) ? 'A' : 'a';

observedFreq[ciphertext[i] - base]++;

totalLetters++;

}

}

// Calculate chi-squared statistic for each possible key

for (int shift = 0; shift < ALPHABET\_SIZE; ++shift) {

char key[ALPHABET\_SIZE];

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

key[i] = (i + shift) % ALPHABET\_SIZE;

}

// Decrypt ciphertext with the current key

char plaintext[256];

decrypt(ciphertext, key, plaintext);

// Calculate chi-squared statistic

float chiSquared = calculateChiSquared(observedFreq);

// Display the decrypted plaintext and chi-squared value

printf("Key Shift: %d, Plaintext: %s, Chi-Squared: %.2f\n", shift, plaintext, chiSquared);

}

}

int main() {

const char \*ciphertext = "Uifsf jt b tfdsfu djqifs xjuipvu uif mfuufs!";

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

int topPlaintexts;

printf("Enter the number of top possible plaintexts to display: ");

scanf("%d", &topPlaintexts);

letterFrequencyAttack(ciphertext, topPlaintexts);

return 0;

}

OUTPUT:

/tmp/vZQqUgQGMX.o

Ciphertext: Uifsf jt b tfdsfu djqifs xjuipvu uif mfuufs!

Enter the number of top possible plaintexts to display: 2

Key Shift: 0, Plaintext: Ntttt tt t tttttt tttttt ttttttt ttt tttttt!, Chi-Squared: 159.34

Key Shift: 1, Plaintext: Mssss ss s ssssss ssssss sssssss sss ssssss!, Chi-Squared: 159.34

Key Shift: 2, Plaintext: Lrrrr rr r rrrrrr rrrrrr rrrrrrr rrr rrrrrr!, Chi-Squared: 159.34

Key Shift: 3, Plaintext: Kqqqq qq q qqqqqq qqqqqq qqqqqqq qqq qqqqqq!, Chi-Squared: 159.34

Key Shift: 4, Plaintext: Jpppp pp p pppppp pppppp ppppppp ppp pppppp!, Chi-Squared: 159.34

Key Shift: 5, Plaintext: Ioooo oo o oooooo oooooo ooooooo ooo oooooo!, Chi-Squared: 159.34

Key Shift: 6, Plaintext: Hnnnn nn n nnnnnn nnnnnn nnnnnnn nnn nnnnnn!, Chi-Squared: 159.34

Key Shift: 7, Plaintext: Gmmmm mm m mmmmmm mmmmmm mmmmmmm mmm mmmmmm!, Chi-Squared: 159.34

Key Shift: 8, Plaintext: Fllll ll l llllll llllll lllllll lll llllll!, Chi-Squared: 159.34

Key Shift: 9, Plaintext: Ekkkk kk k kkkkkk kkkkkk kkkkkkk kkk kkkkkk!, Chi-Squared: 159.34

Key Shift: 10, Plaintext: Djjjj jj j jjjjjj jjjjjj jjjjjjj jjj jjjjjj!, Chi-Squared: 159.34

Key Shift: 11, Plaintext: Ciiii ii i iiiiii iiiiii iiiiiii iii iiiiii!, Chi-Squared: 159.34

Key Shift: 12, Plaintext: Bhhhh hh h hhhhhh hhhhhh hhhhhhh hhh hhhhhh!, Chi-Squared: 159.34

Key Shift: 13, Plaintext: Agggg gg g gggggg gggggg ggggggg ggg gggggg!, Chi-Squared: 159.34

Key Shift: 14, Plaintext: Zffff ff f ffffff ffffff fffffff fff ffffff!, Chi-Squared: 159.34

Key Shift: 15, Plaintext: Yeeee ee e eeeeee eeeeee eeeeeee eee eeeeee!, Chi-Squared: 159.34

Key Shift: 16, Plaintext: Xdddd dd d dddddd dddddd ddddddd ddd dddddd!, Chi-Squared: 159.34

Key Shift: 17, Plaintext: Wcccc cc c cccccc cccccc ccccccc ccc cccccc!, Chi-Squared: 159.34

Key Shift: 18, Plaintext: Vbbbb bb b bbbbbb bbbbbb bbbbbbb bbb bbbbbb!, Chi-Squared: 159.34

Key Shift: 19, Plaintext: Uaaaa aa a aaaaaa aaaaaa aaaaaaa aaa aaaaaa!, Chi-Squared: 159.34

Key Shift: 20, Plaintext: Tzzzz zz z zzzzzz zzzzzz zzzzzzz zzz zzzzzz!, Chi-Squared: 159.34

Key Shift: 21, Plaintext: Syyyy yy y yyyyyy yyyyyy yyyyyyy yyy yyyyyy!, Chi-Squared: 159.34

Key Shift: 22, Plaintext: Rxxxx xx x xxxxxx xxxxxx xxxxxxx xxx xxxxxx!, Chi-Squared: 159.34

Key Shift: 23, Plaintext: Qwwww ww w wwwwww wwwwww wwwwwww www wwwwww!, Chi-Squared: 159.34

Key Shift: 24, Plaintext: Pvvvv vv v vvvvvv vvvvvv vvvvvvv vvv vvvvvv!, Chi-Squared: 159.34

Key Shift: 25, Plaintext: Ouuuu uu u uuuuuu uuuuuu uuuuuuu uuu uuuuuu!, Chi-Squared: 159.34  
  
  
  
38)

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define MAX\_SIZE 10

// Function to calculate 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; // Invalid inverse

}

// Function to encrypt plaintext using Hill cipher

void encrypt(int keyMatrix[MAX\_SIZE][MAX\_SIZE], int plaintextMatrix[MAX\_SIZE], int resultMatrix[MAX\_SIZE], int size) {

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

resultMatrix[i] = 0;

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

resultMatrix[i] += keyMatrix[i][j] \* plaintextMatrix[j];

}

resultMatrix[i] %= 26; // Modulo 26 arithmetic for English alphabets

}

}

// Function to decrypt ciphertext using Hill cipher

void decrypt(int keyMatrix[MAX\_SIZE][MAX\_SIZE], int ciphertextMatrix[MAX\_SIZE], int resultMatrix[MAX\_SIZE], int size) {

int determinant = 0;

int adjointMatrix[MAX\_SIZE][MAX\_SIZE];

// Calculating the determinant

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

determinant += (keyMatrix[0][i] \* (keyMatrix[1][(i + 1) % size] \* keyMatrix[2][(i + 2) % size] - keyMatrix[1][(i + 2) % size] \* keyMatrix[2][(i + 1) % size]));

}

determinant %= 26;

// Calculating the adjoint matrix

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

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

adjointMatrix[j][i] = (((keyMatrix[(i + 1) % size][(j + 1) % size] \* keyMatrix[(i + 2) % size][(j + 2) % size]) - (keyMatrix[(i + 1) % size][(j + 2) % size] \* keyMatrix[(i + 2) % size][(j + 1) % size])) % 26);

if (adjointMatrix[j][i] < 0) {

adjointMatrix[j][i] += 26;

}

}

}

int inverse = modInverse(determinant, 26);

if (inverse == -1) {

printf("Invalid key. Determinant has no modular inverse.\n");

return;

}

// Calculating the decrypted matrix by multiplying with the modular inverse

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

resultMatrix[i] = 0;

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

resultMatrix[i] += (adjointMatrix[i][j] \* inverse \* ciphertextMatrix[j]);

}

resultMatrix[i] = (resultMatrix[i] % 26 + 26) % 26; // Ensure positive values

}

}

// Function to display the matrix

void displayMatrix(int matrix[MAX\_SIZE], int size) {

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

printf("%c", (matrix[i] + 'A')); // Convert numerical values to characters (assuming capitals)

}

printf("\n");

}

int main() {

int keyMatrix[MAX\_SIZE][MAX\_SIZE];

int plaintextMatrix[MAX\_SIZE];

int encryptedMatrix[MAX\_SIZE];

int decryptedMatrix[MAX\_SIZE];

int size;

printf("Enter the size of the key matrix: ");

scanf("%d", &size);

printf("Enter the key matrix elements:\n");

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

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

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

}

}

printf("Enter the plaintext matrix elements: ");

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

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

}

encrypt(keyMatrix, plaintextMatrix, encryptedMatrix, size);

printf("Encrypted text: ");

displayMatrix(encryptedMatrix, size);

decrypt(keyMatrix, encryptedMatrix, decryptedMatrix, size);

printf("Decrypted text: ");

displayMatrix(decryptedMatrix, size);

return 0;

}

OUTPUT:

/tmp/vZQqUgQGMX.o

Enter the size of the key matrix: 2

Enter the key matrix elements:

1

2

3

4

Enter the plaintext matrix elements: 5

6

Encrypted text: RN

Invalid key. Determinant has no modular inverse.

Decrypted text: Aa  
  
39)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define ALPHABET\_SIZE 26

// Function to perform a letter frequency attack on an additive cipher

void frequencyAttack(const char \*ciphertext, int topPlaintexts) {

int frequency[ALPHABET\_SIZE] = {0};

// Count the frequency of each letter in the ciphertext

for (size\_t i = 0; i < strlen(ciphertext); i++) {

char c = ciphertext[i];

if (c >= 'A' && c <= 'Z') {

frequency[c - 'A']++;

} else if (c >= 'a' && c <= 'z') {

frequency[c - 'a']++;

}

}

// Find the most common letter in the English language (E)

int maxFrequencyIndex = 0;

for (int i = 1; i < ALPHABET\_SIZE; i++) {

if (frequency[i] > frequency[maxFrequencyIndex]) {

maxFrequencyIndex = i;

}

}

// Calculate the likely key (distance from E)

int key = (maxFrequencyIndex + ALPHABET\_SIZE - ('E' - 'A')) % ALPHABET\_SIZE;

// Display the most likely plaintexts using the calculated key

printf("Top %d possible plaintexts:\n", topPlaintexts);

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

printf("%d. ", i + 1);

// Decrypt the ciphertext using the key

for (size\_t j = 0; j < strlen(ciphertext); j++) {

char c = ciphertext[j];

if (c >= 'A' && c <= 'Z') {

printf("%c", 'A' + (c - 'A' - key + ALPHABET\_SIZE) % ALPHABET\_SIZE);

} else if (c >= 'a' && c <= 'z') {

printf("%c", 'a' + (c - 'a' - key + ALPHABET\_SIZE) % ALPHABET\_SIZE);

} else {

printf("%c", c);

}

}

printf("\n");

}

}

int main() {

char ciphertext[1000];

int topPlaintexts;

// Input ciphertext from the user

printf("Enter the ciphertext: ");

fgets(ciphertext, sizeof(ciphertext), stdin);

strtok(ciphertext, "\n"); // Remove newline character

// Input the number of top plaintexts to display

printf("Enter the number of top plaintexts to display: ");

scanf("%d", &topPlaintexts);

// Perform letter frequency attack

frequencyAttack(ciphertext, topPlaintexts);

return 0;

}

OUTPUT:

/tmp/vZQqUgQGMX.o

Enter the ciphertext: cryptography

Enter the number of top plaintexts to display: 2

Top 2 possible plaintexts:

1. rgneidvgpewn

2. rgneidvgpewn  
  
  
40)

#include <stdio.h> #include <stdint.h> #include <string.h> #define BLOCK\_SIZE 8 uint8\_t sdes\_encrypt(uint8\_t block, uint16\_t key) { return block ^ (key & 0xFF); } uint8\_t sdes\_decrypt(uint8\_t block, uint16\_t key) { return block ^ (key & 0xFF); } void ctr\_mode\_encrypt(const uint8\_t \*plaintext, uint8\_t \*ciphertext, size\_t len, uint16\_t key, uint8\_t counter) { for (size\_t i = 0; i < len; i++) { uint8\_t encrypted\_counter = sdes\_encrypt(counter, key); ciphertext[i] = plaintext[i] ^ encrypted\_counter; counter++; } } void ctr\_mode\_decrypt(const uint8\_t \*ciphertext, uint8\_t \*plaintext, size\_t len, uint16\_t key, uint8\_t counter) { ctr\_mode\_encrypt(ciphertext, plaintext, len, key, counter); } int main() { uint8\_t plaintext[] = { 0x01, 0x02, 0x04 }; uint16\_t key = 0x1FD; uint8\_t counter = 0x00; size\_t len = sizeof(plaintext) / sizeof(plaintext[0]); uint8\_t ciphertext[len]; uint8\_t decryptedtext[len]; ctr\_mode\_encrypt(plaintext, ciphertext, len, key, counter); printf("Ciphertext: "); for (size\_t i = 0; i < len; i++) { printf("%02X ", ciphertext[i]); } printf("\n"); ctr\_mode\_decrypt(ciphertext, decryptedtext, len, key, counter); printf("Decrypted Plaintext: "); for (size\_t i = 0; i < len; i++) { printf("%02X ", decryptedtext[i]); } printf("\n"); return 0; } OUTPUT: Ciphertext: FC FE FB Decrypted Plaintext: 01 02 04

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

#include <stdio.h>

#include <stdint.h>

#include <string.h>

#define BLOCK\_SIZE 8

uint8\_t sdes\_encrypt(uint8\_t block, uint16\_t key) {

return block ^ (key & 0xFF);

}

uint8\_t sdes\_decrypt(uint8\_t block, uint16\_t key) {

return block ^ (key & 0xFF);

}

void ctr\_mode\_encrypt(const uint8\_t \*plaintext, uint8\_t \*ciphertext, size\_t len, uint16\_t key, uint8\_t counter) {

for (size\_t i = 0; i < len; i++) {

uint8\_t encrypted\_counter = sdes\_encrypt(counter, key);

ciphertext[i] = plaintext[i] ^ encrypted\_counter;

counter++;

}

}

void ctr\_mode\_decrypt(const uint8\_t \*ciphertext, uint8\_t \*plaintext, size\_t len, uint16\_t key, uint8\_t counter) {

ctr\_mode\_encrypt(ciphertext, plaintext, len, key, counter);

}

int main() {

uint8\_t plaintext[] = { 0x01, 0x02, 0x04 };

uint16\_t key = 0x1FD;

uint8\_t counter = 0x00;

size\_t len = sizeof(plaintext) / sizeof(plaintext[0]);

uint8\_t ciphertext[len];

uint8\_t decryptedtext[len];

ctr\_mode\_encrypt(plaintext, ciphertext, len, key, counter);

printf("Ciphertext: ");

for (size\_t i = 0; i < len; i++) {

printf("%02X ", ciphertext[i]);

}

printf("\n");

ctr\_mode\_decrypt(ciphertext, decryptedtext, len, key, counter);

printf("Decrypted Plaintext: ");

for (size\_t i = 0; i < len; i++) {

printf("%02X ", decryptedtext[i]);

}

printf("\n");

return 0;

}

OUTPUT:

Ciphertext: FC FE FB

Decrypted Plaintext: 01 02 04