Program:

#include<bits/stdc++.h>

using namespace std;

int main() {

char str[]="Hello World";

int i,len;

char str1[11];

char str2[11];

len = strlen(str);

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

str1[i]=str[i] & 127;

cout<<str1[i];

}

cout<<endl;

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

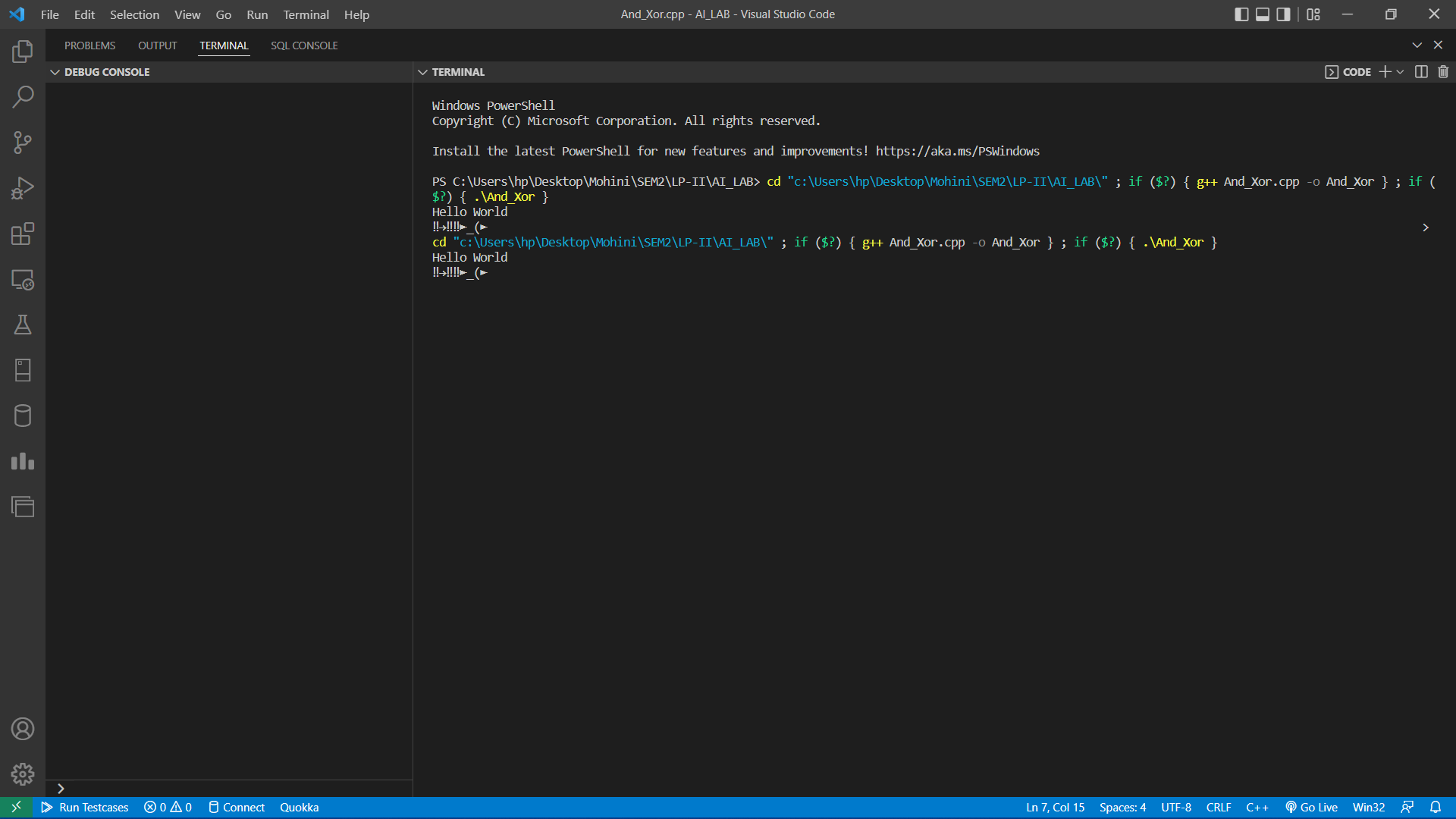
str2[i]=str[i]^127;

cout<<str2[i];

}

cout<<endl;

}



Program:

#include<bits/stdc++.h>

using namespace std;

string const key = "HACK";

map<int,int> keyMap;

void setPermutationOrder()

{

for(int i=0; i < key.length(); i++)

{

keyMap[key[i]] = i;

}

}

string encryptMessage(string msg)

{

int row,col,j;

string cipher = "";

col = key.length();

row = msg.length()/col;

if (msg.length() % col)

row += 1;

char matrix[row][col];

for (int i=0,k=0; i < row; i++)

{

for (int j=0; j<col; )

{

if(msg[k] == '\0')

{

matrix[i][j] = '\_';

j++;

}

if( isalpha(msg[k]) || msg[k]==' ')

{

matrix[i][j] = msg[k];

j++;

}

k++;

}

}

for (map<int,int>::iterator ii = keyMap.begin(); ii!=keyMap.end(); ++ii)

{

j=ii->second;

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

{

if( isalpha(matrix[i][j]) || matrix[i][j]==' ' || matrix[i][j]=='\_')

cipher += matrix[i][j];

}

}

return cipher;

}

string decryptMessage(string cipher)

{

int col = key.length();

int row = cipher.length()/col;

char cipherMat[row][col];

for (int j=0,k=0; j<col; j++)

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

cipherMat[i][j] = cipher[k++];

int index = 0;

for( map<int,int>::iterator ii=keyMap.begin(); ii!=keyMap.end(); ++ii)

ii->second = index++;

char decCipher[row][col];

map<int,int>::iterator ii=keyMap.begin();

int k = 0;

for (int l=0,j; key[l]!='\0'; k++)

{

j = keyMap[key[l++]];

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

{

decCipher[i][k]=cipherMat[i][j];

}

}

string msg = "";

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

{

for(int j=0; j<col; j++)

{

if(decCipher[i][j] != '\_')

msg += decCipher[i][j];

}

}

return msg;

}

int main(void)

{

string msg = "Geeks for Geeks";

setPermutationOrder();

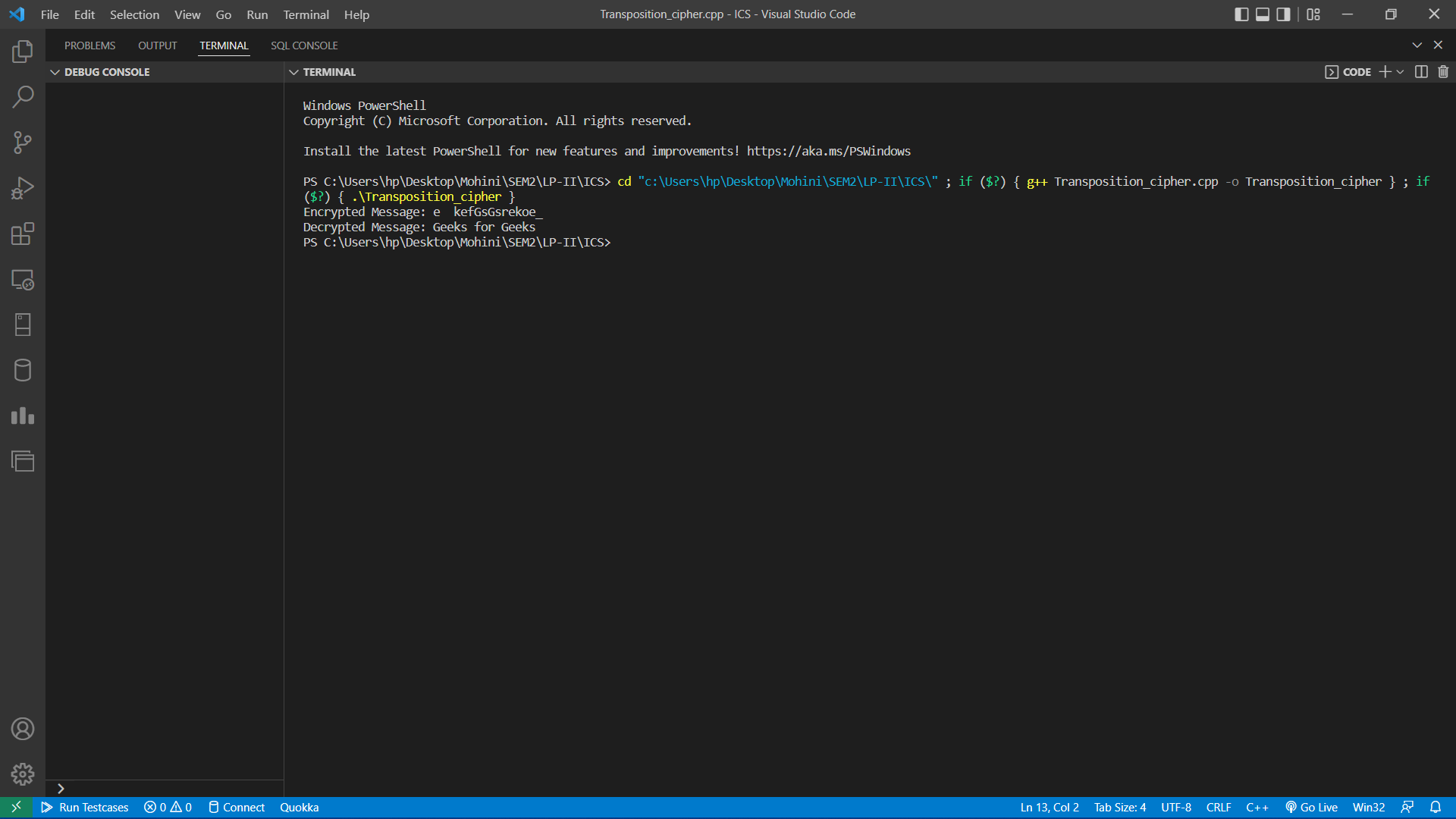
string cipher = encryptMessage(msg);

cout << "Encrypted Message: " << cipher << endl;

cout << "Decrypted Message: " << decryptMessage(cipher) << endl;

return 0;

}



Program:

# Hexadecimal to binary conversion

def hex2bin(s):

mp = {'0' : "0000",

'1' : "0001",

'2' : "0010",

'3' : "0011",

'4' : "0100",

'5' : "0101",

'6' : "0110",

'7' : "0111",

'8' : "1000",

'9' : "1001",

'A' : "1010",

'B' : "1011",

'C' : "1100",

'D' : "1101",

'E' : "1110",

'F' : "1111" }

bin = ""

for i in range(len(s)):

bin = bin + mp[s[i]]

return bin

# Binary to hexadecimal conversion

def bin2hex(s):

mp = {"0000" : '0',

"0001" : '1',

"0010" : '2',

"0011" : '3',

"0100" : '4',

"0101" : '5',

"0110" : '6',

"0111" : '7',

"1000" : '8',

"1001" : '9',

"1010" : 'A',

"1011" : 'B',

"1100" : 'C',

"1101" : 'D',

"1110" : 'E',

"1111" : 'F' }

hex = ""

for i in range(0,len(s),4):

ch = ""

ch = ch + s[i]

ch = ch + s[i + 1]

ch = ch + s[i + 2]

ch = ch + s[i + 3]

hex = hex + mp[ch]

return hex

# Binary to decimal conversion

def bin2dec(binary):

binary1 = binary

decimal, i, n = 0, 0, 0

while(binary != 0):

dec = binary % 10

decimal = decimal + dec \* pow(2, i)

binary = binary//10

i += 1

return decimal

# Decimal to binary conversion

def dec2bin(num):

res = bin(num).replace("0b", "")

if(len(res)%4 != 0):

div = len(res) / 4

div = int(div)

counter =(4 \* (div + 1)) - len(res)

for i in range(0, counter):

res = '0' + res

return res

# Permute function to rearrange the bits

def permute(k, arr, n):

permutation = ""

for i in range(0, n):

permutation = permutation + k[arr[i] - 1]

return permutation

# shifting the bits towards left by nth shifts

def shift\_left(k, nth\_shifts):

s = ""

for i in range(nth\_shifts):

for j in range(1,len(k)):

s = s + k[j]

s = s + k[0]

k = s

s = ""

return k

# calculating xow of two strings of binary number a and b

def xor(a, b):

ans = ""

for i in range(len(a)):

if a[i] == b[i]:

ans = ans + "0"

else:

ans = ans + "1"

return ans

# Table of Position of 64 bits at initial level: Initial Permutation Table

initial\_perm = [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]

# Expansion D-box Table

exp\_d = [32, 1 , 2 , 3 , 4 , 5 , 4 , 5,

6 , 7 , 8 , 9 , 8 , 9 , 10, 11,

12, 13, 12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21, 20, 21,

22, 23, 24, 25, 24, 25, 26, 27,

28, 29, 28, 29, 30, 31, 32, 1 ]

# Straight Permutation Table

per = [ 16, 7, 20, 21,

29, 12, 28, 17,

1, 15, 23, 26,

5, 18, 31, 10,

2, 8, 24, 14,

32, 27, 3, 9,

19, 13, 30, 6,

22, 11, 4, 25 ]

# S-box Table

sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],

[ 0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],

[ 4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],

[15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 ]],

[[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

[3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

[0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

[13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 ]],

[ [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],

[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],

[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12 ]],

[ [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],

[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],

[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],

[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14] ],

[ [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],

[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],

[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],

[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3 ]],

[ [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13] ],

[ [4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12] ],

[ [13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],

[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],

[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],

[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11] ] ]

# Final Permutation Table

final\_perm = [ 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 ]

def encrypt(pt, rkb, rk):

pt = hex2bin(pt)

# Initial Permutation

pt = permute(pt, initial\_perm, 64)

print("After initial permutation", bin2hex(pt))

# Splitting

left = pt[0:32]

right = pt[32:64]

for i in range(0, 16):

# Expansion D-box: Expanding the 32 bits data into 48 bits

right\_expanded = permute(right, exp\_d, 48)

# XOR RoundKey[i] and right\_expanded

xor\_x = xor(right\_expanded, rkb[i])

# S-boxex: substituting the value from s-box table by calculating row and column

sbox\_str = ""

for j in range(0, 8):

row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))

col = bin2dec(int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3] + xor\_x[j \* 6 + 4]))

val = sbox[j][row][col]

sbox\_str = sbox\_str + dec2bin(val)

# Straight D-box: After substituting rearranging the bits

sbox\_str = permute(sbox\_str, per, 32)

# XOR left and sbox\_str

result = xor(left, sbox\_str)

left = result

# Swapper

if(i != 15):

left, right = right, left

print("Round ", i + 1, " ", bin2hex(left), " ", bin2hex(right), " ", rk[i])

# Combination

combine = left + right

# Final permutation: final rearranging of bits to get cipher text

cipher\_text = permute(combine, final\_perm, 64)

return cipher\_text

pt = "123456ABCD132536"

key = "AABB09182736CCDD"

# Key generation

# --hex to binary

key = hex2bin(key)

# --parity bit drop table

keyp = [57, 49, 41, 33, 25, 17, 9,

1, 58, 50, 42, 34, 26, 18,

10, 2, 59, 51, 43, 35, 27,

19, 11, 3, 60, 52, 44, 36,

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

7, 62, 54, 46, 38, 30, 22,

14, 6, 61, 53, 45, 37, 29,

21, 13, 5, 28, 20, 12, 4 ]

# getting 56 bit key from 64 bit using the parity bits

key = permute(key, keyp, 56)

# Number of bit shifts

shift\_table = [1, 1, 2, 2,

2, 2, 2, 2,

1, 2, 2, 2,

2, 2, 2, 1 ]

# Key- Compression Table : Compression of key from 56 bits to 48 bits

key\_comp = [14, 17, 11, 24, 1, 5,

3, 28, 15, 6, 21, 10,

23, 19, 12, 4, 26, 8,

16, 7, 27, 20, 13, 2,

41, 52, 31, 37, 47, 55,

30, 40, 51, 45, 33, 48,

44, 49, 39, 56, 34, 53,

46, 42, 50, 36, 29, 32 ]

# Splitting

left = key[0:28] # rkb for RoundKeys in binary

right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []

rk = []

for i in range(0, 16):

# Shifting the bits by nth shifts by checking from shift table

left = shift\_left(left, shift\_table[i])

right = shift\_left(right, shift\_table[i])

# Combination of left and right string

combine\_str = left + right

# Compression of key from 56 to 48 bits

round\_key = permute(combine\_str, key\_comp, 48)

rkb.append(round\_key)

rk.append(bin2hex(round\_key))

print("Encryption")

cipher\_text = bin2hex(encrypt(pt, rkb, rk))

print("Cipher Text : ",cipher\_text)

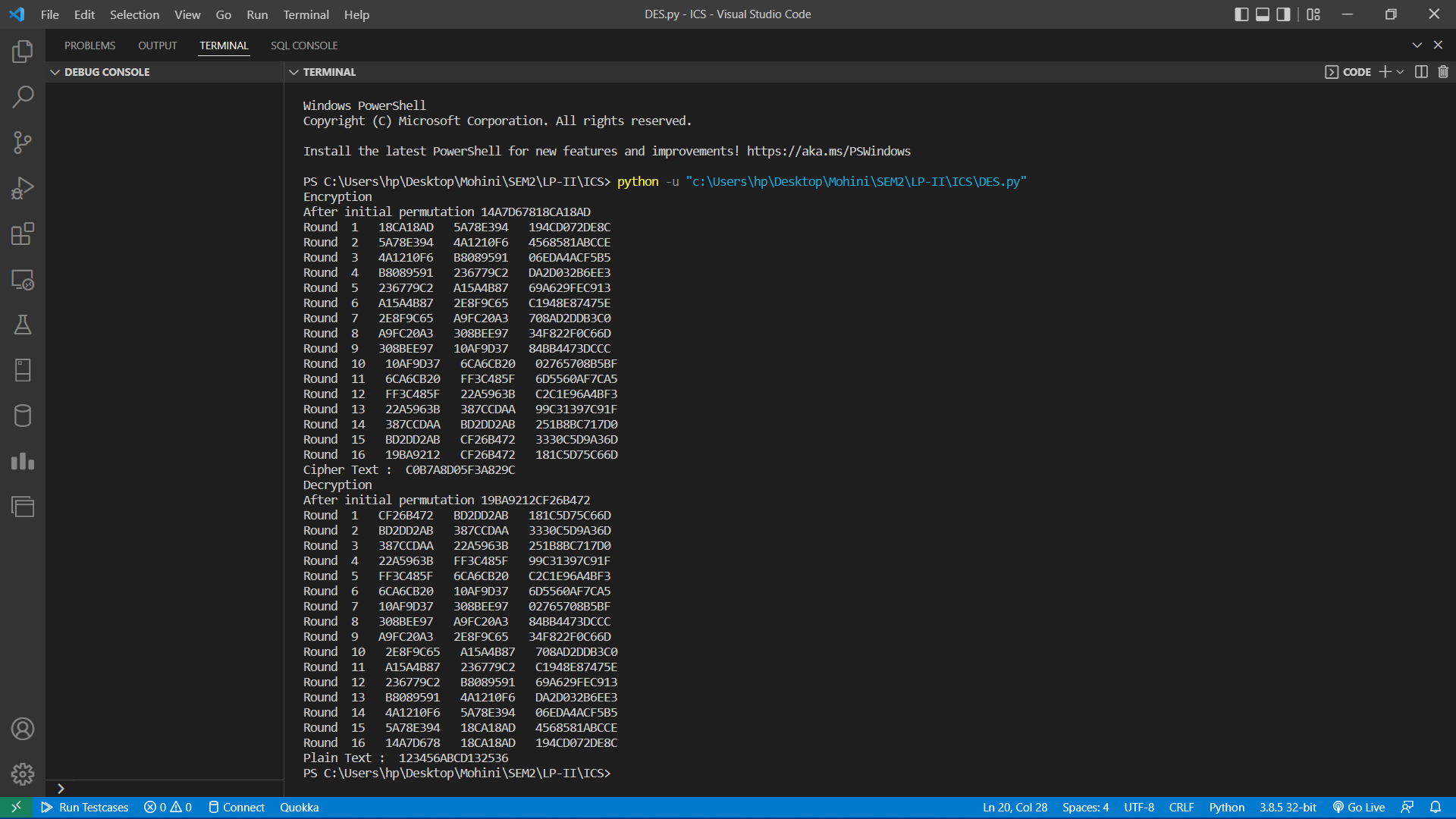
print("Decryption")

rkb\_rev = rkb[::-1]

rk\_rev = rk[::-1]

text = bin2hex(encrypt(cipher\_text, rkb\_rev, rk\_rev))

print("Plain Text : ",text)



Program:

#include<iostream>

#include<math.h>

using namespace std;

// find gcd

int gcd(int a, int b) {

int t;

while(1) {

t= a%b;

if(t==0)

return b;

a = b;

b= t;

}

}

int main() {

//2 random prime numbers

double p = 13;

double q = 11;

double n=p\*q;//calculate n

double track;

double phi= (p-1)\*(q-1);//calculate phi

//public key

//e stands for encrypt

double e=7;

//for checking that 1 < e < phi(n) and gcd(e, phi(n)) = 1; i.e., e and phi(n) are coprime.

while(e<phi) {

track = gcd(e,phi);

if(track==1)

break;

else

e++;

}

//private key

//d stands for decrypt

//choosing d such that it satisfies d\*e = 1 mod phi

double d1=1/e;

double d=fmod(d1,phi);

double message = 9;

double c = pow(message,e); //encrypt the message

double m = pow(c,d);

c=fmod(c,n);

m=fmod(m,n);

cout<<"Original Message = "<<message;

cout<<"\n"<<"p = "<<p;

cout<<"\n"<<"q = "<<q;

cout<<"\n"<<"n = pq = "<<n;

cout<<"\n"<<"phi = "<<phi;

cout<<"\n"<<"e = "<<e;

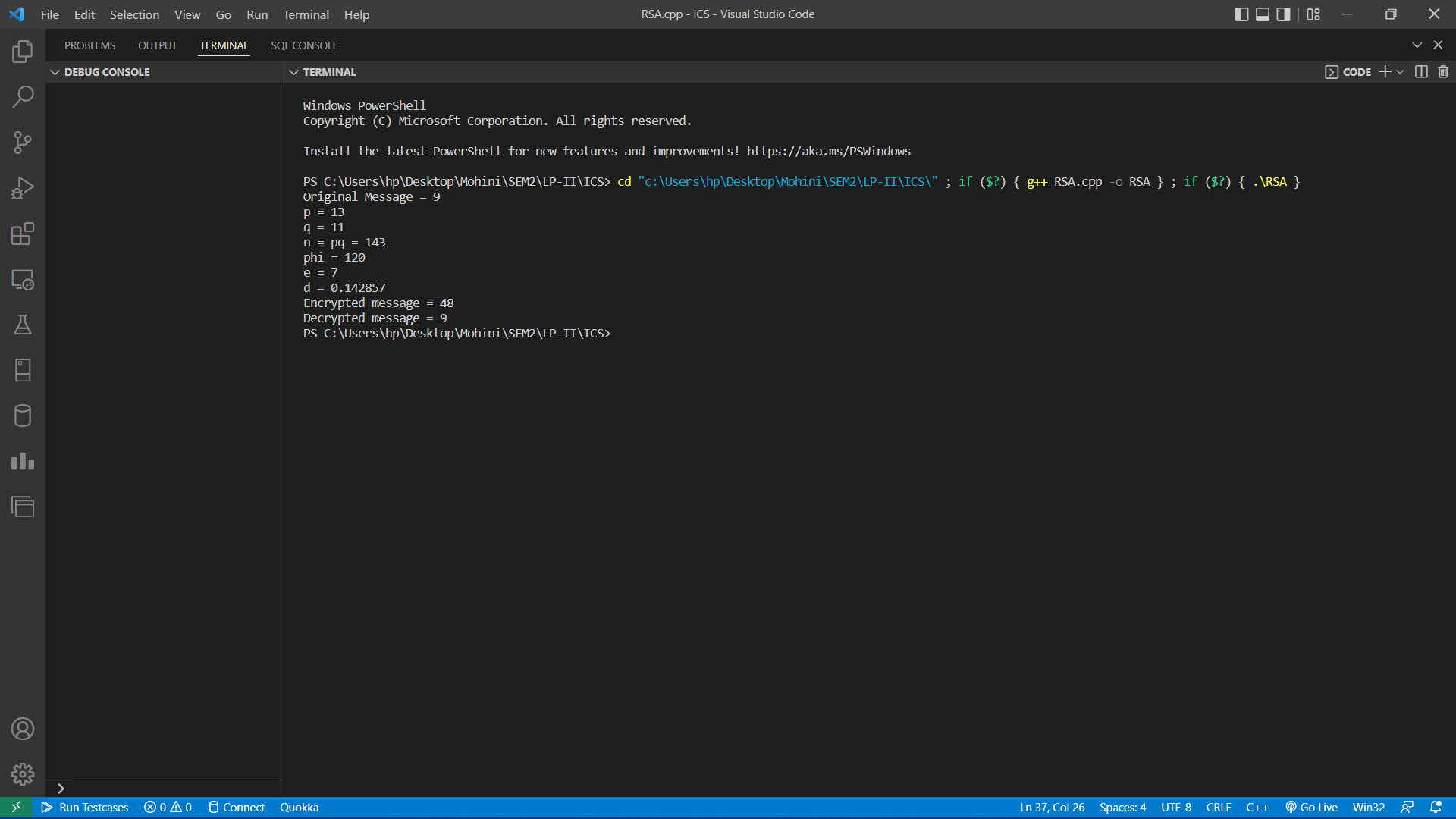
cout<<"\n"<<"d = "<<d;

cout<<"\n"<<"Encrypted message = "<<c;

cout<<"\n"<<"Decrypted message = "<<m;

return 0;

}



Program:

Encrypt.cpp

/\* encrypt.cpp

\* Performs encryption using AES 128-bit

\* @author Cecelia Wisniewska

\*/

#include <iostream>

#include <cstring>

#include <fstream>

#include <sstream>

#include "structures.h"

using namespace std;

/\* Serves as the initial round during encryption

\* AddRoundKey is simply an XOR of a 128-bit block with the 128-bit key.

\*/

void AddRoundKey(unsigned char \* state, unsigned char \* roundKey) {

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

state[i] ^= roundKey[i];

}

}

/\* Perform substitution to each of the 16 bytes

\* Uses S-box as lookup table

\*/

void SubBytes(unsigned char \* state) {

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

state[i] = s[state[i]];

}

}

// Shift left, adds diffusion

void ShiftRows(unsigned char \* state) {

unsigned char tmp[16];

/\* Column 1 \*/

tmp[0] = state[0];

tmp[1] = state[5];

tmp[2] = state[10];

tmp[3] = state[15];

/\* Column 2 \*/

tmp[4] = state[4];

tmp[5] = state[9];

tmp[6] = state[14];

tmp[7] = state[3];

/\* Column 3 \*/

tmp[8] = state[8];

tmp[9] = state[13];

tmp[10] = state[2];

tmp[11] = state[7];

/\* Column 4 \*/

tmp[12] = state[12];

tmp[13] = state[1];

tmp[14] = state[6];

tmp[15] = state[11];

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

state[i] = tmp[i];

}

}

/\* MixColumns uses mul2, mul3 look-up tables

\* Source of diffusion

\*/

void MixColumns(unsigned char \* state) {

unsigned char tmp[16];

tmp[0] = (unsigned char) mul2[state[0]] ^ mul3[state[1]] ^ state[2] ^ state[3];

tmp[1] = (unsigned char) state[0] ^ mul2[state[1]] ^ mul3[state[2]] ^ state[3];

tmp[2] = (unsigned char) state[0] ^ state[1] ^ mul2[state[2]] ^ mul3[state[3]];

tmp[3] = (unsigned char) mul3[state[0]] ^ state[1] ^ state[2] ^ mul2[state[3]];

tmp[4] = (unsigned char)mul2[state[4]] ^ mul3[state[5]] ^ state[6] ^ state[7];

tmp[5] = (unsigned char)state[4] ^ mul2[state[5]] ^ mul3[state[6]] ^ state[7];

tmp[6] = (unsigned char)state[4] ^ state[5] ^ mul2[state[6]] ^ mul3[state[7]];

tmp[7] = (unsigned char)mul3[state[4]] ^ state[5] ^ state[6] ^ mul2[state[7]];

tmp[8] = (unsigned char)mul2[state[8]] ^ mul3[state[9]] ^ state[10] ^ state[11];

tmp[9] = (unsigned char)state[8] ^ mul2[state[9]] ^ mul3[state[10]] ^ state[11];

tmp[10] = (unsigned char)state[8] ^ state[9] ^ mul2[state[10]] ^ mul3[state[11]];

tmp[11] = (unsigned char)mul3[state[8]] ^ state[9] ^ state[10] ^ mul2[state[11]];

tmp[12] = (unsigned char)mul2[state[12]] ^ mul3[state[13]] ^ state[14] ^ state[15];

tmp[13] = (unsigned char)state[12] ^ mul2[state[13]] ^ mul3[state[14]] ^ state[15];

tmp[14] = (unsigned char)state[12] ^ state[13] ^ mul2[state[14]] ^ mul3[state[15]];

tmp[15] = (unsigned char)mul3[state[12]] ^ state[13] ^ state[14] ^ mul2[state[15]];

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

state[i] = tmp[i];

}

}

/\* Each round operates on 128 bits at a time

\* The number of rounds is defined in AESEncrypt()

\*/

void Round(unsigned char \* state, unsigned char \* key) {

SubBytes(state);

ShiftRows(state);

MixColumns(state);

AddRoundKey(state, key);

}

// Same as Round() except it doesn't mix columns

void FinalRound(unsigned char \* state, unsigned char \* key) {

SubBytes(state);

ShiftRows(state);

AddRoundKey(state, key);

}

/\* The AES encryption function

\* Organizes the confusion and diffusion steps into one function

\*/

void AESEncrypt(unsigned char \* message, unsigned char \* expandedKey, unsigned char \* encryptedMessage) {

unsigned char state[16]; // Stores the first 16 bytes of original message

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

state[i] = message[i];

}

int numberOfRounds = 9;

AddRoundKey(state, expandedKey); // Initial round

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

Round(state, expandedKey + (16 \* (i+1)));

}

FinalRound(state, expandedKey + 160);

// Copy encrypted state to buffer

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

encryptedMessage[i] = state[i];

}

}

int main() {

cout << "=============================" << endl;

cout << " 128-bit AES Encryption Tool " << endl;

cout << "=============================" << endl;

char message[1024];

cout << "Enter the message to encrypt: ";

cin.getline(message, sizeof(message));

cout << message << endl;

// Pad message to 16 bytes

int originalLen = strlen((const char \*)message);

int paddedMessageLen = originalLen;

if ((paddedMessageLen % 16) != 0) {

paddedMessageLen = (paddedMessageLen / 16 + 1) \* 16;

}

unsigned char \* paddedMessage = new unsigned char[paddedMessageLen];

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

if (i >= originalLen) {

paddedMessage[i] = 0;

}

else {

paddedMessage[i] = message[i];

}

}

unsigned char \* encryptedMessage = new unsigned char[paddedMessageLen];

string str;

ifstream infile;

infile.open("keyfile", ios::in | ios::binary);

if (infile.is\_open())

{

getline(infile, str); // The first line of file should be the key

infile.close();

}

else cout << "Unable to open file";

istringstream hex\_chars\_stream(str);

unsigned char key[16];

int i = 0;

unsigned int c;

while (hex\_chars\_stream >> hex >> c)

{

key[i] = c;

i++;

}

unsigned char expandedKey[176];

KeyExpansion(key, expandedKey);

for (int i = 0; i < paddedMessageLen; i += 16) {

AESEncrypt(paddedMessage+i, expandedKey, encryptedMessage+i);

}

cout << "Encrypted message in hex:" << endl;

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

cout << hex << (int) encryptedMessage[i];

cout << " ";

}

cout << endl;

// Write the encrypted string out to file "message.aes"

ofstream outfile;

outfile.open("message.aes", ios::out | ios::binary);

if (outfile.is\_open())

{

outfile << encryptedMessage;

outfile.close();

cout << "Wrote encrypted message to file message.aes" << endl;

}

else cout << "Unable to open file";

// Free memory

delete[] paddedMessage;

delete[] encryptedMessage;

return 0;

}

Decrypt.cpp

/\* decrypt.cpp

\* Performs decryption using AES 128-bit

\* @author Cecelia Wisniewska

\*/

#include <iostream>

#include <cstring>

#include <fstream>

#include <sstream>

#include "structures.h"

using namespace std;

/\* Used in Round() and serves as the final round during decryption

\* SubRoundKey is simply an XOR of a 128-bit block with the 128-bit key.

\* So basically does the same as AddRoundKey in the encryption

\*/

void SubRoundKey(unsigned char \* state, unsigned char \* roundKey) {

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

state[i] ^= roundKey[i];

}

}

/\* InverseMixColumns uses mul9, mul11, mul13, mul14 look-up tables

\* Unmixes the columns by reversing the effect of MixColumns in encryption

\*/

void InverseMixColumns(unsigned char \* state) {

unsigned char tmp[16];

tmp[0] = (unsigned char)mul14[state[0]] ^ mul11[state[1]] ^ mul13[state[2]] ^ mul9[state[3]];

tmp[1] = (unsigned char)mul9[state[0]] ^ mul14[state[1]] ^ mul11[state[2]] ^ mul13[state[3]];

tmp[2] = (unsigned char)mul13[state[0]] ^ mul9[state[1]] ^ mul14[state[2]] ^ mul11[state[3]];

tmp[3] = (unsigned char)mul11[state[0]] ^ mul13[state[1]] ^ mul9[state[2]] ^ mul14[state[3]];

tmp[4] = (unsigned char)mul14[state[4]] ^ mul11[state[5]] ^ mul13[state[6]] ^ mul9[state[7]];

tmp[5] = (unsigned char)mul9[state[4]] ^ mul14[state[5]] ^ mul11[state[6]] ^ mul13[state[7]];

tmp[6] = (unsigned char)mul13[state[4]] ^ mul9[state[5]] ^ mul14[state[6]] ^ mul11[state[7]];

tmp[7] = (unsigned char)mul11[state[4]] ^ mul13[state[5]] ^ mul9[state[6]] ^ mul14[state[7]];

tmp[8] = (unsigned char)mul14[state[8]] ^ mul11[state[9]] ^ mul13[state[10]] ^ mul9[state[11]];

tmp[9] = (unsigned char)mul9[state[8]] ^ mul14[state[9]] ^ mul11[state[10]] ^ mul13[state[11]];

tmp[10] = (unsigned char)mul13[state[8]] ^ mul9[state[9]] ^ mul14[state[10]] ^ mul11[state[11]];

tmp[11] = (unsigned char)mul11[state[8]] ^ mul13[state[9]] ^ mul9[state[10]] ^ mul14[state[11]];

tmp[12] = (unsigned char)mul14[state[12]] ^ mul11[state[13]] ^ mul13[state[14]] ^ mul9[state[15]];

tmp[13] = (unsigned char)mul9[state[12]] ^ mul14[state[13]] ^ mul11[state[14]] ^ mul13[state[15]];

tmp[14] = (unsigned char)mul13[state[12]] ^ mul9[state[13]] ^ mul14[state[14]] ^ mul11[state[15]];

tmp[15] = (unsigned char)mul11[state[12]] ^ mul13[state[13]] ^ mul9[state[14]] ^ mul14[state[15]];

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

state[i] = tmp[i];

}

}

// Shifts rows right (rather than left) for decryption

void ShiftRows(unsigned char \* state) {

unsigned char tmp[16];

/\* Column 1 \*/

tmp[0] = state[0];

tmp[1] = state[13];

tmp[2] = state[10];

tmp[3] = state[7];

/\* Column 2 \*/

tmp[4] = state[4];

tmp[5] = state[1];

tmp[6] = state[14];

tmp[7] = state[11];

/\* Column 3 \*/

tmp[8] = state[8];

tmp[9] = state[5];

tmp[10] = state[2];

tmp[11] = state[15];

/\* Column 4 \*/

tmp[12] = state[12];

tmp[13] = state[9];

tmp[14] = state[6];

tmp[15] = state[3];

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

state[i] = tmp[i];

}

}

/\* Perform substitution to each of the 16 bytes

\* Uses inverse S-box as lookup table

\*/

void SubBytes(unsigned char \* state) {

for (int i = 0; i < 16; i++) { // Perform substitution to each of the 16 bytes

state[i] = inv\_s[state[i]];

}

}

/\* Each round operates on 128 bits at a time

\* The number of rounds is defined in AESDecrypt()

\* Not surprisingly, the steps are the encryption steps but reversed

\*/

void Round(unsigned char \* state, unsigned char \* key) {

SubRoundKey(state, key);

InverseMixColumns(state);

ShiftRows(state);

SubBytes(state);

}

// Same as Round() but no InverseMixColumns

void InitialRound(unsigned char \* state, unsigned char \* key) {

SubRoundKey(state, key);

ShiftRows(state);

SubBytes(state);

}

/\* The AES decryption function

\* Organizes all the decryption steps into one function

\*/

void AESDecrypt(unsigned char \* encryptedMessage, unsigned char \* expandedKey, unsigned char \* decryptedMessage)

{

unsigned char state[16]; // Stores the first 16 bytes of encrypted message

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

state[i] = encryptedMessage[i];

}

InitialRound(state, expandedKey+160);

int numberOfRounds = 9;

for (int i = 8; i >= 0; i--) {

Round(state, expandedKey + (16 \* (i + 1)));

}

SubRoundKey(state, expandedKey); // Final round

// Copy decrypted state to buffer

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

decryptedMessage[i] = state[i];

}

}

int main() {

cout << "=============================" << endl;

cout << " 128-bit AES Decryption Tool " << endl;

cout << "=============================" << endl;

// Read in the message from message.aes

string msgstr;

ifstream infile;

infile.open("message.aes", ios::in | ios::binary);

if (infile.is\_open())

{

getline(infile, msgstr); // The first line of file is the message

cout << "Read in encrypted message from message.aes" << endl;

infile.close();

}

else cout << "Unable to open file";

char \* msg = new char[msgstr.size()+1];

strcpy(msg, msgstr.c\_str());

int n = strlen((const char\*)msg);

unsigned char \* encryptedMessage = new unsigned char[n];

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

encryptedMessage[i] = (unsigned char)msg[i];

}

// Free memory

delete[] msg;

// Read in the key

string keystr;

ifstream keyfile;

keyfile.open("keyfile", ios::in | ios::binary);

if (keyfile.is\_open())

{

getline(keyfile, keystr); // The first line of file should be the key

cout << "Read in the 128-bit key from keyfile" << endl;

keyfile.close();

}

else cout << "Unable to open file";

istringstream hex\_chars\_stream(keystr);

unsigned char key[16];

int i = 0;

unsigned int c;

while (hex\_chars\_stream >> hex >> c)

{

key[i] = c;

i++;

}

unsigned char expandedKey[176];

KeyExpansion(key, expandedKey);

int messageLen = strlen((const char \*)encryptedMessage);

unsigned char \* decryptedMessage = new unsigned char[messageLen];

for (int i = 0; i < messageLen; i += 16) {

AESDecrypt(encryptedMessage + i, expandedKey, decryptedMessage + i);

}

cout << "Decrypted message in hex:" << endl;

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

cout << hex << (int)decryptedMessage[i];

cout << " ";

}

cout << endl;

cout << "Decrypted message: ";

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

cout << decryptedMessage[i];

}

cout << endl;

return 0;

}

Structures.h

/\* "structures.h" defines the look-up tables and KeyExpansion function

\* used in encrypt.cpp and decrypt.cpp

\*/

#ifndef STRUCTURES\_H

#define STRUCTURES\_H

// Encryption: Forward Rijndael S-box

unsigned char s[256] =

{

0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,

0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,

0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,

0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,

0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,

0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,

0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,

0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,

0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,

0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,

0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,

0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,

0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,

0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,

0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,

0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16

};

// Encryption: Multiply by 2 for MixColumns

unsigned char mul2[] =

{

0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,0x10,0x12,0x14,0x16,0x18,0x1a,0x1c,0x1e,

0x20,0x22,0x24,0x26,0x28,0x2a,0x2c,0x2e,0x30,0x32,0x34,0x36,0x38,0x3a,0x3c,0x3e,

0x40,0x42,0x44,0x46,0x48,0x4a,0x4c,0x4e,0x50,0x52,0x54,0x56,0x58,0x5a,0x5c,0x5e,

0x60,0x62,0x64,0x66,0x68,0x6a,0x6c,0x6e,0x70,0x72,0x74,0x76,0x78,0x7a,0x7c,0x7e,

0x80,0x82,0x84,0x86,0x88,0x8a,0x8c,0x8e,0x90,0x92,0x94,0x96,0x98,0x9a,0x9c,0x9e,

0xa0,0xa2,0xa4,0xa6,0xa8,0xaa,0xac,0xae,0xb0,0xb2,0xb4,0xb6,0xb8,0xba,0xbc,0xbe,

0xc0,0xc2,0xc4,0xc6,0xc8,0xca,0xcc,0xce,0xd0,0xd2,0xd4,0xd6,0xd8,0xda,0xdc,0xde,

0xe0,0xe2,0xe4,0xe6,0xe8,0xea,0xec,0xee,0xf0,0xf2,0xf4,0xf6,0xf8,0xfa,0xfc,0xfe,

0x1b,0x19,0x1f,0x1d,0x13,0x11,0x17,0x15,0x0b,0x09,0x0f,0x0d,0x03,0x01,0x07,0x05,

0x3b,0x39,0x3f,0x3d,0x33,0x31,0x37,0x35,0x2b,0x29,0x2f,0x2d,0x23,0x21,0x27,0x25,

0x5b,0x59,0x5f,0x5d,0x53,0x51,0x57,0x55,0x4b,0x49,0x4f,0x4d,0x43,0x41,0x47,0x45,

0x7b,0x79,0x7f,0x7d,0x73,0x71,0x77,0x75,0x6b,0x69,0x6f,0x6d,0x63,0x61,0x67,0x65,

0x9b,0x99,0x9f,0x9d,0x93,0x91,0x97,0x95,0x8b,0x89,0x8f,0x8d,0x83,0x81,0x87,0x85,

0xbb,0xb9,0xbf,0xbd,0xb3,0xb1,0xb7,0xb5,0xab,0xa9,0xaf,0xad,0xa3,0xa1,0xa7,0xa5,

0xdb,0xd9,0xdf,0xdd,0xd3,0xd1,0xd7,0xd5,0xcb,0xc9,0xcf,0xcd,0xc3,0xc1,0xc7,0xc5,

0xfb,0xf9,0xff,0xfd,0xf3,0xf1,0xf7,0xf5,0xeb,0xe9,0xef,0xed,0xe3,0xe1,0xe7,0xe5

};

// Encryption: Multiply by 3 for MixColumns

unsigned char mul3[] =

{

0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0x1b,0x1e,0x1d,0x14,0x17,0x12,0x11,

0x30,0x33,0x36,0x35,0x3c,0x3f,0x3a,0x39,0x28,0x2b,0x2e,0x2d,0x24,0x27,0x22,0x21,

0x60,0x63,0x66,0x65,0x6c,0x6f,0x6a,0x69,0x78,0x7b,0x7e,0x7d,0x74,0x77,0x72,0x71,

0x50,0x53,0x56,0x55,0x5c,0x5f,0x5a,0x59,0x48,0x4b,0x4e,0x4d,0x44,0x47,0x42,0x41,

0xc0,0xc3,0xc6,0xc5,0xcc,0xcf,0xca,0xc9,0xd8,0xdb,0xde,0xdd,0xd4,0xd7,0xd2,0xd1,

0xf0,0xf3,0xf6,0xf5,0xfc,0xff,0xfa,0xf9,0xe8,0xeb,0xee,0xed,0xe4,0xe7,0xe2,0xe1,

0xa0,0xa3,0xa6,0xa5,0xac,0xaf,0xaa,0xa9,0xb8,0xbb,0xbe,0xbd,0xb4,0xb7,0xb2,0xb1,

0x90,0x93,0x96,0x95,0x9c,0x9f,0x9a,0x99,0x88,0x8b,0x8e,0x8d,0x84,0x87,0x82,0x81,

0x9b,0x98,0x9d,0x9e,0x97,0x94,0x91,0x92,0x83,0x80,0x85,0x86,0x8f,0x8c,0x89,0x8a,

0xab,0xa8,0xad,0xae,0xa7,0xa4,0xa1,0xa2,0xb3,0xb0,0xb5,0xb6,0xbf,0xbc,0xb9,0xba,

0xfb,0xf8,0xfd,0xfe,0xf7,0xf4,0xf1,0xf2,0xe3,0xe0,0xe5,0xe6,0xef,0xec,0xe9,0xea,

0xcb,0xc8,0xcd,0xce,0xc7,0xc4,0xc1,0xc2,0xd3,0xd0,0xd5,0xd6,0xdf,0xdc,0xd9,0xda,

0x5b,0x58,0x5d,0x5e,0x57,0x54,0x51,0x52,0x43,0x40,0x45,0x46,0x4f,0x4c,0x49,0x4a,

0x6b,0x68,0x6d,0x6e,0x67,0x64,0x61,0x62,0x73,0x70,0x75,0x76,0x7f,0x7c,0x79,0x7a,

0x3b,0x38,0x3d,0x3e,0x37,0x34,0x31,0x32,0x23,0x20,0x25,0x26,0x2f,0x2c,0x29,0x2a,

0x0b,0x08,0x0d,0x0e,0x07,0x04,0x01,0x02,0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a

};

// Used in KeyExpansion

unsigned char rcon[256] = {

0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,

0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,

0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,

0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,

0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,

0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,

0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,

0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,

0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,

0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,

0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,

0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,

0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,

0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,

0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,

0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d

};

// Decryption: Inverse Rijndael S-box

unsigned char inv\_s[256] =

{

0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,

0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,

0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,

0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,

0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,

0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,

0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,

0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,

0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,

0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,

0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,

0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,

0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,

0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,

0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,

0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D

};

// Decryption: Multiply by 9 for InverseMixColumns

unsigned char mul9[256] =

{

0x00,0x09,0x12,0x1b,0x24,0x2d,0x36,0x3f,0x48,0x41,0x5a,0x53,0x6c,0x65,0x7e,0x77,

0x90,0x99,0x82,0x8b,0xb4,0xbd,0xa6,0xaf,0xd8,0xd1,0xca,0xc3,0xfc,0xf5,0xee,0xe7,

0x3b,0x32,0x29,0x20,0x1f,0x16,0x0d,0x04,0x73,0x7a,0x61,0x68,0x57,0x5e,0x45,0x4c,

0xab,0xa2,0xb9,0xb0,0x8f,0x86,0x9d,0x94,0xe3,0xea,0xf1,0xf8,0xc7,0xce,0xd5,0xdc,

0x76,0x7f,0x64,0x6d,0x52,0x5b,0x40,0x49,0x3e,0x37,0x2c,0x25,0x1a,0x13,0x08,0x01,

0xe6,0xef,0xf4,0xfd,0xc2,0xcb,0xd0,0xd9,0xae,0xa7,0xbc,0xb5,0x8a,0x83,0x98,0x91,

0x4d,0x44,0x5f,0x56,0x69,0x60,0x7b,0x72,0x05,0x0c,0x17,0x1e,0x21,0x28,0x33,0x3a,

0xdd,0xd4,0xcf,0xc6,0xf9,0xf0,0xeb,0xe2,0x95,0x9c,0x87,0x8e,0xb1,0xb8,0xa3,0xaa,

0xec,0xe5,0xfe,0xf7,0xc8,0xc1,0xda,0xd3,0xa4,0xad,0xb6,0xbf,0x80,0x89,0x92,0x9b,

0x7c,0x75,0x6e,0x67,0x58,0x51,0x4a,0x43,0x34,0x3d,0x26,0x2f,0x10,0x19,0x02,0x0b,

0xd7,0xde,0xc5,0xcc,0xf3,0xfa,0xe1,0xe8,0x9f,0x96,0x8d,0x84,0xbb,0xb2,0xa9,0xa0,

0x47,0x4e,0x55,0x5c,0x63,0x6a,0x71,0x78,0x0f,0x06,0x1d,0x14,0x2b,0x22,0x39,0x30,

0x9a,0x93,0x88,0x81,0xbe,0xb7,0xac,0xa5,0xd2,0xdb,0xc0,0xc9,0xf6,0xff,0xe4,0xed,

0x0a,0x03,0x18,0x11,0x2e,0x27,0x3c,0x35,0x42,0x4b,0x50,0x59,0x66,0x6f,0x74,0x7d,

0xa1,0xa8,0xb3,0xba,0x85,0x8c,0x97,0x9e,0xe9,0xe0,0xfb,0xf2,0xcd,0xc4,0xdf,0xd6,

0x31,0x38,0x23,0x2a,0x15,0x1c,0x07,0x0e,0x79,0x70,0x6b,0x62,0x5d,0x54,0x4f,0x46

};

// Decryption: Multiply by 11 for InverseMixColumns

unsigned char mul11[256] =

{

0x00,0x0b,0x16,0x1d,0x2c,0x27,0x3a,0x31,0x58,0x53,0x4e,0x45,0x74,0x7f,0x62,0x69,

0xb0,0xbb,0xa6,0xad,0x9c,0x97,0x8a,0x81,0xe8,0xe3,0xfe,0xf5,0xc4,0xcf,0xd2,0xd9,

0x7b,0x70,0x6d,0x66,0x57,0x5c,0x41,0x4a,0x23,0x28,0x35,0x3e,0x0f,0x04,0x19,0x12,

0xcb,0xc0,0xdd,0xd6,0xe7,0xec,0xf1,0xfa,0x93,0x98,0x85,0x8e,0xbf,0xb4,0xa9,0xa2,

0xf6,0xfd,0xe0,0xeb,0xda,0xd1,0xcc,0xc7,0xae,0xa5,0xb8,0xb3,0x82,0x89,0x94,0x9f,

0x46,0x4d,0x50,0x5b,0x6a,0x61,0x7c,0x77,0x1e,0x15,0x08,0x03,0x32,0x39,0x24,0x2f,

0x8d,0x86,0x9b,0x90,0xa1,0xaa,0xb7,0xbc,0xd5,0xde,0xc3,0xc8,0xf9,0xf2,0xef,0xe4,

0x3d,0x36,0x2b,0x20,0x11,0x1a,0x07,0x0c,0x65,0x6e,0x73,0x78,0x49,0x42,0x5f,0x54,

0xf7,0xfc,0xe1,0xea,0xdb,0xd0,0xcd,0xc6,0xaf,0xa4,0xb9,0xb2,0x83,0x88,0x95,0x9e,

0x47,0x4c,0x51,0x5a,0x6b,0x60,0x7d,0x76,0x1f,0x14,0x09,0x02,0x33,0x38,0x25,0x2e,

0x8c,0x87,0x9a,0x91,0xa0,0xab,0xb6,0xbd,0xd4,0xdf,0xc2,0xc9,0xf8,0xf3,0xee,0xe5,

0x3c,0x37,0x2a,0x21,0x10,0x1b,0x06,0x0d,0x64,0x6f,0x72,0x79,0x48,0x43,0x5e,0x55,

0x01,0x0a,0x17,0x1c,0x2d,0x26,0x3b,0x30,0x59,0x52,0x4f,0x44,0x75,0x7e,0x63,0x68,

0xb1,0xba,0xa7,0xac,0x9d,0x96,0x8b,0x80,0xe9,0xe2,0xff,0xf4,0xc5,0xce,0xd3,0xd8,

0x7a,0x71,0x6c,0x67,0x56,0x5d,0x40,0x4b,0x22,0x29,0x34,0x3f,0x0e,0x05,0x18,0x13,

0xca,0xc1,0xdc,0xd7,0xe6,0xed,0xf0,0xfb,0x92,0x99,0x84,0x8f,0xbe,0xb5,0xa8,0xa3

};

// Decryption: Multiply by 13 for InverseMixColumns

unsigned char mul13[256] =

{

0x00,0x0d,0x1a,0x17,0x34,0x39,0x2e,0x23,0x68,0x65,0x72,0x7f,0x5c,0x51,0x46,0x4b,

0xd0,0xdd,0xca,0xc7,0xe4,0xe9,0xfe,0xf3,0xb8,0xb5,0xa2,0xaf,0x8c,0x81,0x96,0x9b,

0xbb,0xb6,0xa1,0xac,0x8f,0x82,0x95,0x98,0xd3,0xde,0xc9,0xc4,0xe7,0xea,0xfd,0xf0,

0x6b,0x66,0x71,0x7c,0x5f,0x52,0x45,0x48,0x03,0x0e,0x19,0x14,0x37,0x3a,0x2d,0x20,

0x6d,0x60,0x77,0x7a,0x59,0x54,0x43,0x4e,0x05,0x08,0x1f,0x12,0x31,0x3c,0x2b,0x26,

0xbd,0xb0,0xa7,0xaa,0x89,0x84,0x93,0x9e,0xd5,0xd8,0xcf,0xc2,0xe1,0xec,0xfb,0xf6,

0xd6,0xdb,0xcc,0xc1,0xe2,0xef,0xf8,0xf5,0xbe,0xb3,0xa4,0xa9,0x8a,0x87,0x90,0x9d,

0x06,0x0b,0x1c,0x11,0x32,0x3f,0x28,0x25,0x6e,0x63,0x74,0x79,0x5a,0x57,0x40,0x4d,

0xda,0xd7,0xc0,0xcd,0xee,0xe3,0xf4,0xf9,0xb2,0xbf,0xa8,0xa5,0x86,0x8b,0x9c,0x91,

0x0a,0x07,0x10,0x1d,0x3e,0x33,0x24,0x29,0x62,0x6f,0x78,0x75,0x56,0x5b,0x4c,0x41,

0x61,0x6c,0x7b,0x76,0x55,0x58,0x4f,0x42,0x09,0x04,0x13,0x1e,0x3d,0x30,0x27,0x2a,

0xb1,0xbc,0xab,0xa6,0x85,0x88,0x9f,0x92,0xd9,0xd4,0xc3,0xce,0xed,0xe0,0xf7,0xfa,

0xb7,0xba,0xad,0xa0,0x83,0x8e,0x99,0x94,0xdf,0xd2,0xc5,0xc8,0xeb,0xe6,0xf1,0xfc,

0x67,0x6a,0x7d,0x70,0x53,0x5e,0x49,0x44,0x0f,0x02,0x15,0x18,0x3b,0x36,0x21,0x2c,

0x0c,0x01,0x16,0x1b,0x38,0x35,0x22,0x2f,0x64,0x69,0x7e,0x73,0x50,0x5d,0x4a,0x47,

0xdc,0xd1,0xc6,0xcb,0xe8,0xe5,0xf2,0xff,0xb4,0xb9,0xae,0xa3,0x80,0x8d,0x9a,0x97

};

// Decryption: Multiply by 14 for InverseMixColumns

unsigned char mul14[256] =

{

0x00,0x0e,0x1c,0x12,0x38,0x36,0x24,0x2a,0x70,0x7e,0x6c,0x62,0x48,0x46,0x54,0x5a,

0xe0,0xee,0xfc,0xf2,0xd8,0xd6,0xc4,0xca,0x90,0x9e,0x8c,0x82,0xa8,0xa6,0xb4,0xba,

0xdb,0xd5,0xc7,0xc9,0xe3,0xed,0xff,0xf1,0xab,0xa5,0xb7,0xb9,0x93,0x9d,0x8f,0x81,

0x3b,0x35,0x27,0x29,0x03,0x0d,0x1f,0x11,0x4b,0x45,0x57,0x59,0x73,0x7d,0x6f,0x61,

0xad,0xa3,0xb1,0xbf,0x95,0x9b,0x89,0x87,0xdd,0xd3,0xc1,0xcf,0xe5,0xeb,0xf9,0xf7,

0x4d,0x43,0x51,0x5f,0x75,0x7b,0x69,0x67,0x3d,0x33,0x21,0x2f,0x05,0x0b,0x19,0x17,

0x76,0x78,0x6a,0x64,0x4e,0x40,0x52,0x5c,0x06,0x08,0x1a,0x14,0x3e,0x30,0x22,0x2c,

0x96,0x98,0x8a,0x84,0xae,0xa0,0xb2,0xbc,0xe6,0xe8,0xfa,0xf4,0xde,0xd0,0xc2,0xcc,

0x41,0x4f,0x5d,0x53,0x79,0x77,0x65,0x6b,0x31,0x3f,0x2d,0x23,0x09,0x07,0x15,0x1b,

0xa1,0xaf,0xbd,0xb3,0x99,0x97,0x85,0x8b,0xd1,0xdf,0xcd,0xc3,0xe9,0xe7,0xf5,0xfb,

0x9a,0x94,0x86,0x88,0xa2,0xac,0xbe,0xb0,0xea,0xe4,0xf6,0xf8,0xd2,0xdc,0xce,0xc0,

0x7a,0x74,0x66,0x68,0x42,0x4c,0x5e,0x50,0x0a,0x04,0x16,0x18,0x32,0x3c,0x2e,0x20,

0xec,0xe2,0xf0,0xfe,0xd4,0xda,0xc8,0xc6,0x9c,0x92,0x80,0x8e,0xa4,0xaa,0xb8,0xb6,

0x0c,0x02,0x10,0x1e,0x34,0x3a,0x28,0x26,0x7c,0x72,0x60,0x6e,0x44,0x4a,0x58,0x56,

0x37,0x39,0x2b,0x25,0x0f,0x01,0x13,0x1d,0x47,0x49,0x5b,0x55,0x7f,0x71,0x63,0x6d,

0xd7,0xd9,0xcb,0xc5,0xef,0xe1,0xf3,0xfd,0xa7,0xa9,0xbb,0xb5,0x9f,0x91,0x83,0x8d

};

// Auxiliary function for KeyExpansion

void KeyExpansionCore(unsigned char \* in, unsigned char i) {

// Rotate left by one byte: shift left

unsigned char t = in[0];

in[0] = in[1];

in[1] = in[2];

in[2] = in[3];

in[3] = t;

// S-box 4 bytes

in[0] = s[in[0]];

in[1] = s[in[1]];

in[2] = s[in[2]];

in[3] = s[in[3]];

// RCon

in[0] ^= rcon[i];

}

/\* The main KeyExpansion function

\* Generates additional keys using the original key

\* Total of 11 128-bit keys generated, including the original

\* Keys are stored one after the other in expandedKeys

\*/

void KeyExpansion(unsigned char inputKey[16], unsigned char expandedKeys[176]) {

// The first 128 bits are the original key

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

expandedKeys[i] = inputKey[i];

}

int bytesGenerated = 16; // Bytes we've generated so far

int rconIteration = 1; // Keeps track of rcon value

unsigned char tmpCore[4]; // Temp storage for core

while (bytesGenerated < 176) {

/\* Read 4 bytes for the core

\* They are the previously generated 4 bytes

\* Initially, these will be the final 4 bytes of the original key

\*/

for (int i = 0; i < 4; i++) {

tmpCore[i] = expandedKeys[i + bytesGenerated - 4];

}

// Perform the core once for each 16 byte key

if (bytesGenerated % 16 == 0) {

KeyExpansionCore(tmpCore, rconIteration++);

}

for (unsigned char a = 0; a < 4; a++) {

expandedKeys[bytesGenerated] = expandedKeys[bytesGenerated - 16] ^ tmpCore[a];

bytesGenerated++;

}

}

}

#endif /\* STRUCTURES\_H \*/

