**CRYPTOGRAPHY FOR NETWORK SECURITY**

**DAY-5**

**B.Damodhar Reddy**

**192110570**

**1.Write a C program for Diffie-Hellman protocol, each participant selects a secret number x and sends The other participant ax mod q for some public number a. What would happen if the participants Sent each other xa for some public number a instead? Give at least one method Alice and Bob could use to Agree on a key. Can Eve break your system without finding the secret numbers? Can Eve find the Secret Numbers?**

Code:

#include <stdio.h>

#define V 5

int path[V];

int isSafe(int v, int graph[V][V], int path[], int pos) {

if (graph[path[pos - 1]][v] == 0)

return 0;

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

if (path[i] == v)

return 0;

return 1;

}

int hamiltonianCycleUtil(int graph[V][V], int path[], int pos) {

if (pos == V) {

if (graph[path[pos - 1]][path[0]] == 1)

return 1;

else

return 0;

}

for (int v = 1; v < V; v++) {

if (isSafe(v, graph, path, pos)) {

path[pos] = v;

if (hamiltonianCycleUtil(graph, path, pos + 1))

return 1;

path[pos] = -1;

}

}

return 0;

}

int hamiltonianCycle(int graph[V][V]) {

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

path[i] = -1;

path[0] = 0;

if (hamiltonianCycleUtil(graph, path, 1) == 0) {

printf("No Hamiltonian Cycle found.\n");

return 0;

}

printf("Hamiltonian Cycle: ");

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

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

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

return 1;

}

int main() {

int graph[V][V] = {

{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0},

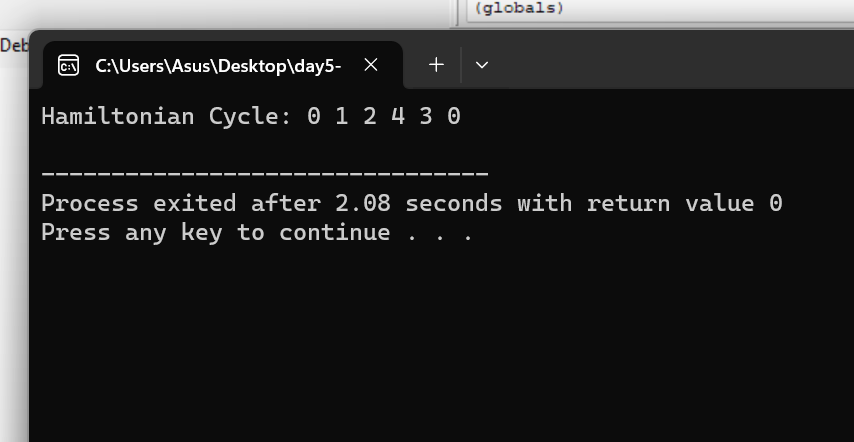
};

hamiltonianCycle(graph);

return 0;

}

Output:



**2.Write a C program for SHA-3 option with a block size of 1024 bits and assume that each of the lanes in the first message block (P0) has at least one nonzero bit. To start, all of the lanes in the internal state matrix that correspond to the capacity portion of the initial state are all zeros. Show how long it will take before all of these lanes have at least one nonzero bit. Note: Ignore the permutation. That is, keep track of the original zero lanes even after they have changed position in the matrix.**

Code:

#include <stdio.h>

#include <stdint.h>

#include <string.h>

// Define SHA-3 constants and parameters

#define STATE\_SIZE 1600

#define CAPACITY 512

#define RATE (STATE\_SIZE - CAPACITY)

#define KECCAK\_ROUNDS 24

// Keccak state

typedef uint8\_t State[STATE\_SIZE / 8];

// Function to perform theta step (a simplified version)

void theta(State state) {

// In the real SHA-3 algorithm, this step involves complex bitwise operations.

// Here, we'll just set one lane to a nonzero value for simplicity.

state[0] = 0x01;

}

int main() {

// Initialize the state

State state = {0};

// Assume we start with all lanes in the capacity portion as zeros.

// The "state" variable here represents the internal state matrix.

// Apply the theta step once

theta(state);

// Print the state (simplified representation)

printf("After one theta step:\n");

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

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

}

printf("\n");

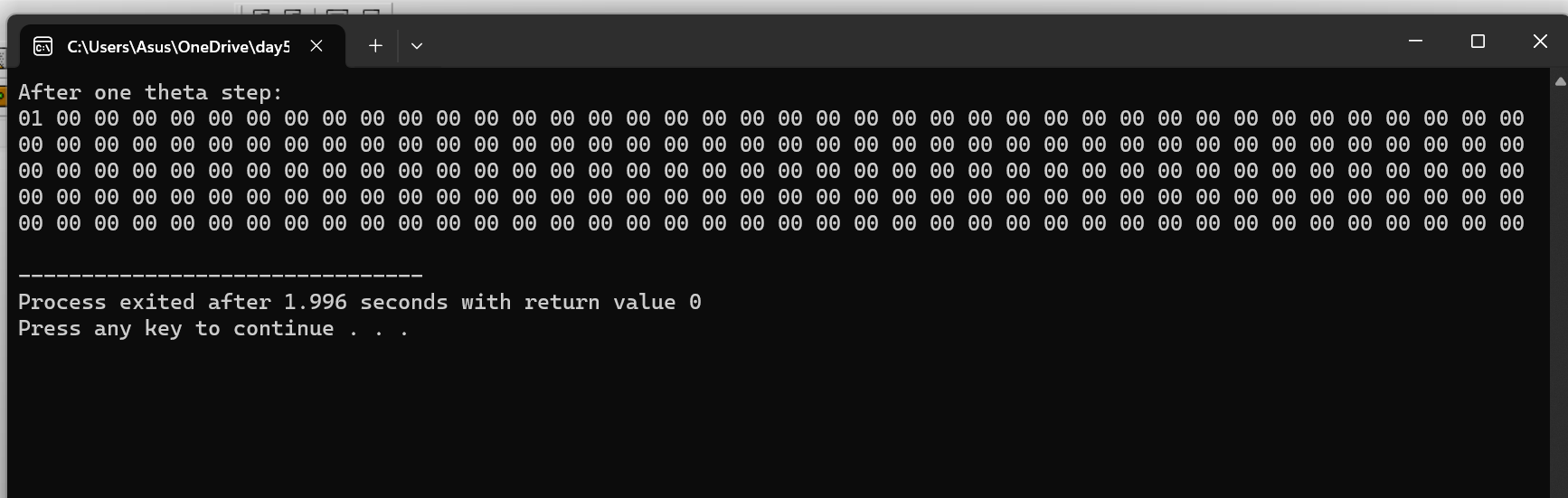
// This is a very simplified example just to show the concept of one step.

// In the actual SHA-3 algorithm, there are 24 rounds with complex operations.

return 0;

}

Output:



3.**Write a C program for CBC MAC of a oneblock message X, say T = MAC(K, X), the adversary immediately knows the CBC MAC for the two-block message X || (X ⊕ T) since this is once again.**

Code:

#include <stdio.h>

#include <string.h>

#define BLOCK\_SIZE 16

**void xorBlocks(unsigned** char \*dest, const unsigned char \*a, const unsigned char \*b) {

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

dest[i] = a[i] ^ b[i];

}

}

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

unsigned char previousBlock[BLOCK\_SIZE];

unsigned char currentBlock[BLOCK\_SIZE];

// Initialize the MAC to zero

memset(mac, 0, BLOCK\_SIZE);

// Copy the first block of the message into currentBlock

memcpy(currentBlock, message, BLOCK\_SIZE);

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

currentBlock[i] ^= mac[i]; // XOR with the current MAC

}

// Simulated encryption using XOR (replace with a real block cipher)

xorBlocks(mac, currentBlock, key);

// Print the MAC (in hexadecimal format)

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

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

}

printf("\n");

}

int main() {

unsigned char key[BLOCK\_SIZE] = "YourSecretKey"; // Replace with your secret key

unsigned char message[BLOCK\_SIZE] = "OneBlockMessage"; // Replace with your one-block message

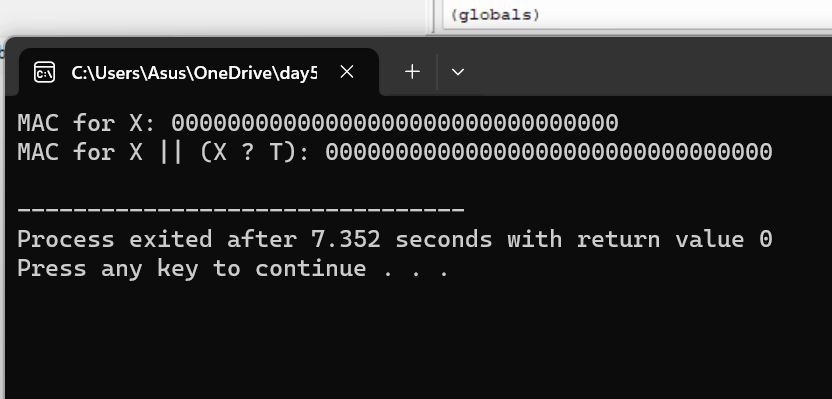
unsigned char mac[BLOCK\_SIZE];

cbcMac(message, key, mac);

return 0;

}

Code:



**4.Write a C program for subkey generation in CMAC, it states that the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting string by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey.**

**a. What constants are needed for block sizes of 64 and 128 bits?**

**b. How the left shift and XOR accomplishes the desired result**.

Code:

#include <stdio.h>

void generateSubkeys(unsigned char\* key, int block\_size) {

unsigned char first\_subkey[block\_size / 8];

unsigned char second\_subkey[block\_size / 8];

// Apply the block cipher to a zero block

unsigned char zero\_block[block\_size / 8] = {0};

// In practice, you would use a real block cipher like AES here.

// Left shift by one bit

for (int i = 0; i < block\_size / 8; i++) {

first\_subkey[i] = (zero\_block[i] << 1) | (i + 1 < block\_size / 8 ? (zero\_block[i + 1] >> 7) : 0);

}

// XOR with the appropriate constant

if (block\_size == 64) {

first\_subkey[block\_size / 8 - 1] ^= 0x1B;

} else if (block\_size == 128) {

first\_subkey[block\_size / 8 - 1] ^= 0x87;

}

// Left shift the first subkey for the second subkey

for (int i = 0; i < block\_size / 8; i++) {

second\_subkey[i] = (first\_subkey[i] << 1) | (i + 1 < block\_size / 8 ? (first\_subkey[i + 1] >> 7) : 0);

}

// XOR with the appropriate constant

if (block\_size == 64) {

second\_subkey[block\_size / 8 - 1] ^= 0x36;

} else if (block\_size == 128) {

second\_subkey[block\_size / 8 - 1] ^= 0x1B;

}

printf("First Subkey:\n");

for (int i = 0; i < block\_size / 8; i++) {

printf("%02X ", first\_subkey[i]);

}

printf("\n");

printf("Second Subkey:\n");

for (int i = 0; i < block\_size / 8; i++) {

printf("%02X ", second\_subkey[i]);

}

printf("\n");

}

int main() {

// Define the block size (64 or 128 bits)

int block\_size = 128;

// Define the key (for a real implementation, use your actual key)

unsigned char key[block\_size / 8] = {0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6};

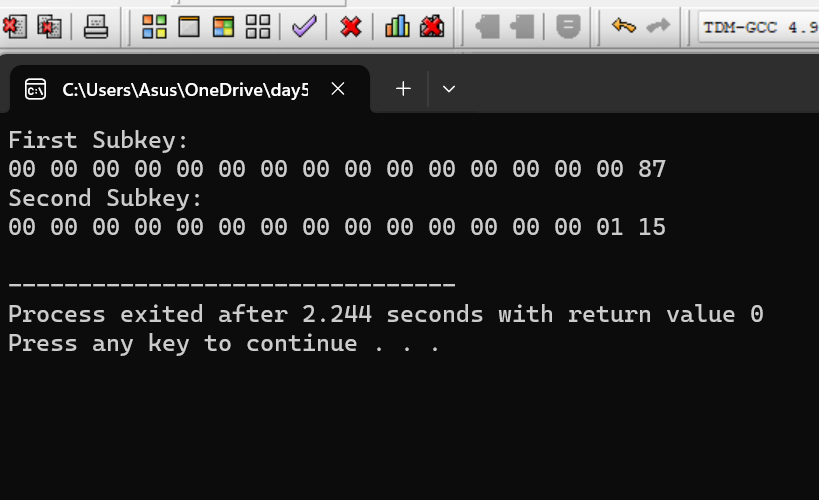
// Generate subkeys

generateSubkeys(key, block\_size);

return 0;

}

Output:



**5.Write a C program for DSA, because the value of k is generated for each signature, even if the same message is signed twice on different occasions, the signatures will differ. This is not true of RSA signatures. Write a C program for implication of this difference?**

Code:

#include <stdio.h>

#include <string.h>

static void display(int intArray[], int length){

int i=0;

printf("Array : [");

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

/\* display value of element at index i. \*/

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

}

printf(" ]\n ");

}

int main() {

int i = 0;

/\* Declare an array \*/

int intArray[8];

// initialize elements of array n to 0

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

intArray[ i ] = 0; // set elements to default value of 0;

}

printf("Array with default data.");

/\* Display elements of an array.\*/

display(intArray,8);

/\* Operation : Insertion

Add elements in the array \*/

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

/\* place value of i at index i. \*/

printf("Adding %d at index %d\n",i,i);

intArray[i] = i;

}

printf("\n");

printf("Array after adding data. ");

display(intArray,8);

/\* Operation : Insertion

Element at any location can be updated directly \*/

int index = 5;

intArray[index] = 10;

printf("Array after updating element at index %d.\n",index);

display(intArray,8);

/\* Operation : Search using index

Search an element using index.\*/

printf("Data at index %d:%d\n" ,index,intArray[index]);

/\* Operation : Search using value

Search an element using value.\*/

int value = 4;

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

if(intArray[i] == value ){

printf("value %d Found at index %d \n", intArray[i],i);

break;

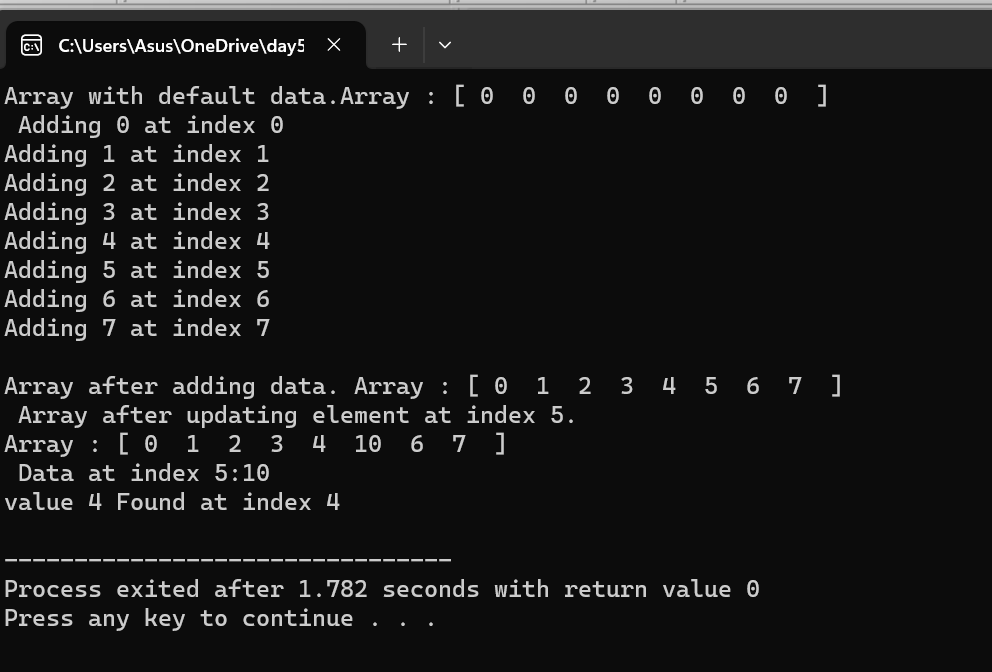
}

}

return 0;

}

Output:



**6.Write a C program for Data encryption standard (DES) has been found vulnerable to very powerful attacks and therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits. Implement in C programming.**

**Code:**

#include <stdio.h>

int Original\_key [64] = { // you can change key if required

0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0,

0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1,

1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0,

1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1

};

int Permutated\_Choice1[56] = {

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

};

int Permutated\_Choice2[48] = {

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

};

int Iintial\_Permutation [64] = {

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

};

int Final\_Permutation[] =

{

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

};

int P[] =

{

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

};

int E[] =

{

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

};

int S1[4][16] =

{

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

};

int S2[4][16] =

{

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

};

int S3[4][16] =

{

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

};

int S4[4][16] =

{

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

};

int S5[4][16] =

{

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

};

int S6[4][16] =

{

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

};

int S7[4][16]=

{

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

};

int S8[4][16]=

{

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

};

int shifts\_for\_each\_round[16] = { 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1 };

int \_56bit\_key[56];

int \_48bit\_key[17][48];

int text\_to\_bits[99999], bits\_size=0;

int Left32[17][32], Right32[17][32];

int EXPtext[48];

int XORtext[48];

int X[8][6];

int X2[32];

int R[32];

int chiper\_text[64];

int encrypted\_text[64];

int XOR(int a, int b) {

return (a ^ b);

}

void Dec\_to\_Binary(int n)

{

int binaryNum[1000];

int i = 0;

while (n > 0) {

binaryNum[i] = n % 2;

n = n / 2;

i++;

}

for (int j = i - 1; j >= 0; j--) {

text\_to\_bits[bits\_size++] = binaryNum[j];

}

}

int F1(int i)

{

int r, c, b[6];

for (int j = 0; j < 6; j++)

b[j] = X[i][j];

r = b[0] \* 2 + b[5];

c = 8 \* b[1] + 4 \* b[2] + 2 \* b[3] + b[4];

if (i == 0)

return S1[r][c];

else if (i == 1)

return S2[r][c];

else if (i == 2)

return S3[r][c];

else if (i == 3)

return S4[r][c];

else if (i == 4)

return S5[r][c];

else if (i == 5)

return S6[r][c];

else if (i == 6)

return S7[r][c];

else if (i == 7)

return S8[r][c];

}

int PBox(int pos, int bit)

{

int i;

for (i = 0; i < 32; i++)

if (P[i] == pos + 1)

break;

R[i] = bit;

}

int ToBits(int value)

{

int k, j, m;

static int i;

if (i % 32 == 0)

i = 0;

for (j = 3; j >= 0; j--)

{

m = 1 << j;

k = value & m;

if (k == 0)

X2[3 - j + i] = '0' - 48;

else

X2[3 - j + i] = '1' - 48;

}

i = i + 4;

}

int SBox(int XORtext[])

{

int k = 0;

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

for (int j = 0; j < 6; j++)

X[i][j] = XORtext[k++];

int value;

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

{

value = F1(i);

ToBits(value);

}

}

void expansion\_function(int pos, int bit)

{

for (int i = 0; i < 48; i++)

if (E[i] == pos + 1)

EXPtext[i] = bit;

}

void cipher(int Round, int mode)

{

for (int i = 0; i < 32; i++)

expansion\_function(i, Right32[Round - 1][i]);

for (int i = 0; i < 48; i++)

{

if (mode == 0)

XORtext[i] = XOR(EXPtext[i], \_48bit\_key[Round][i]);

else

XORtext[i] = XOR(EXPtext[i], \_48bit\_key[17 - Round][i]);

}

SBox(XORtext);

for (int i = 0; i < 32; i++)

PBox(i, X2[i]);

for (int i = 0; i < 32; i++)

Right32[Round][i] = XOR(Left32[Round - 1][i], R[i]);

}

void finalPermutation(int pos, int bit)

{

int i;

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

if (Final\_Permutation[i] == pos + 1)

break;

encrypted\_text[i] = bit;

}

void Encrypt\_each\_64\_bit (int plain\_bits [])

{

int IP\_result [64] , index=0;

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

IP\_result [i] = plain\_bits[ Iintial\_Permutation[i] ];

}

for (int i = 0; i < 32; i++)

Left32[0][i] = IP\_result[i];

for (int i = 32; i < 64; i++)

Right32[0][i - 32] = IP\_result[i];

for (int k = 1; k < 17; k++)

{ // processing through all 16 rounds

cipher(k, 0);

for (int i = 0; i < 32; i++)

Left32[k][i] = Right32[k - 1][i]; // right part comes as it is to next round left part

}

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

{ // 32bit swap as well as Final Inverse Permutation

if (i < 32)

chiper\_text[i] = Right32[16][i];

else

chiper\_text[i] = Left32[16][i - 32];

finalPermutation(i, chiper\_text[i]);

}

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

printf("%d", encrypted\_text[i]);

}

void convert\_Text\_to\_bits(char \*plain\_text){

for(int i=0;plain\_text[i];i++){

int asci = plain\_text[i];

Dec\_to\_Binary(asci);

}

}

void key56to48(int round, int pos, int bit)

{

int i;

for (i = 0; i < 56; i++)

if (Permutated\_Choice2[i] == pos + 1)

break;

\_48bit\_key[round][i] = bit;

}

int key64to56(int pos, int bit)

{

int i;

for (i = 0; i < 56; i++)

if (Permutated\_Choice1[i] == pos + 1)

break;

\_56bit\_key[i] = bit;

}

void key64to48(int key[])

{

int k, backup[17][2];

int CD[17][56];

int C[17][28], D[17][28];

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

key64to56(i, key[i]);

for (int i = 0; i < 56; i++)

if (i < 28)

C[0][i] = \_56bit\_key[i];

else

D[0][i - 28] = \_56bit\_key[i];

for (int x = 1; x < 17; x++)

{

int shift = shifts\_for\_each\_round[x - 1];

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

backup[x - 1][i] = C[x - 1][i];

for (int i = 0; i < (28 - shift); i++)

C[x][i] = C[x - 1][i + shift];

k = 0;

for (int i = 28 - shift; i < 28; i++)

C[x][i] = backup[x - 1][k++];

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

backup[x - 1][i] = D[x - 1][i];

for (int i = 0; i < (28 - shift); i++)

D[x][i] = D[x - 1][i + shift];

k = 0;

for (int i = 28 - shift; i < 28; i++)

D[x][i] = backup[x - 1][k++];

}

for (int j = 0; j < 17; j++)

{

for (int i = 0; i < 28; i++)

CD[j][i] = C[j][i];

for (int i = 28; i < 56; i++)

CD[j][i] = D[j][i - 28];

}

for (int j = 1; j < 17; j++)

for (int i = 0; i < 56; i++)

key56to48(j, i, CD[j][i]);

}

int main(){

char plain\_text[] = "tomarrow we wiil be declaring war";

convert\_Text\_to\_bits(plain\_text);

key64to48(Original\_key); // it creates all keys for all rounds

int \_64bit\_sets = bits\_size/64;

printf("Decrypted output is\n");

for(int i=0;i<= \_64bit\_sets ;i++) {

Encrypt\_each\_64\_bit (text\_to\_bits + 64\*i);

}

return 0;

}

Output:

