

**AHSANULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY**

Department of Computer Science and Engineering

Program: Bachelor of Science in Computer Science and Engineering

Course Code: CSE 4174

Course Title: Cyber Security Lab

Academic Semester: Spring 2023

Assignment Topic: Data Encryption Standard (DES)

Submitted on: December 23, 2023

Submitted by

Name: Yasir Arafah Prince

Student ID: 20200104042

Lab Section: A2

**Question 1:** Data Encryption Standard (DES) is a symmetric key encryption approach. It

has several modes. Two such modes are ECB (Electronic Code Book) and CBC

(Cipher Block Chaining).

a. Between ECB and CBC modes, which mode do you think is more secure?

Justify your answer with proper explanation.

b. Write a program in C/C++/Java that takes a plaintext and a key as inputs

and performs encryption and decryption with the DES mode of your

answer from question a.

**Solution:**

1. CBC (Cipher Block Chaining) is considered to be more secure than ECB (Electronic Codebook) since it conceals data patterns by chaining the encryption process. Because identical ciphertext blocks are produced from similar plaintext blocks, ECB is vulnerable to pattern analysis. CBC, on the other hand, uses the preceding block's ciphertext to alter each plaintext block before encrypting it. By ensuring that comparable plaintext blocks encrypt in different ways, this dramatically increases security by masking patterns.
2. The C++ code is given below

**Code:**

#include <iostream>

#include <string>

#include <vector>

#include <bitset>

using namespace std;

const int PC\_1\_TABLE[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

};

const int SHIFT\_SCHEDULE[16] =

{

1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1

};

const int PC\_2\_TABLE[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

};

const int IP\_TABLE[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

};

const int E\_BIT\_SELECTION\_TABLE[48] =

{

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

};

const int S\_BOXES[8][4][16] =

{

// S1

{

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

},

// S2

{

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

},

// S3

{

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

},

// S4

{

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

},

// S5

{

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

},

// S6

{

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

},

// S7

{

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

},

// S8

{

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

}

};

const int PERMUTATION\_TABLE[32] =

{

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

};

const int INVERSE\_INITIAL\_PERMUTATION[64] =

{

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

};

string initial\_permutation(const string& input)

{

string permuted;

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

permuted += input[IP\_TABLE[i] - 1];

}

return permuted;

}

pair<string, string> PC\_1(const string& key)

{

string permutedKey;

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

{

permutedKey += key[PC\_1\_TABLE[i] - 1];

}

string C = permutedKey.substr(0, 28);

string D = permutedKey.substr(28, 28);

return make\_pair(C, D);

}

string leftCircularShift(const string& half, int shifts)

{

string shifted = half.substr(shifts) + half.substr(0, shifts);

return shifted;

}

string PC\_2(const string& C, const string& D)

{

string combined = C + D;

string subkey;

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

{

subkey += combined[PC\_2\_TABLE[i] - 1];

}

return subkey;

}

vector<string> performRounds(pair<string, string>& halves)

{

vector<string> roundKeys(16);

cout << "Initial halves after PC-1:" << endl;

cout << "28 bit C: " << halves.first << endl;

cout << "28 bit D: " << halves.second << endl;

cout << endl;

for (int Round = 0; Round < 16; Round++)

{

halves.first = leftCircularShift(halves.first, SHIFT\_SCHEDULE[Round]);

halves.second = leftCircularShift(halves.second, SHIFT\_SCHEDULE[Round]);

cout << "Round " << (Round + 1) << " shifted values:" << endl;

cout << "C" << (Round + 1) << ": " << halves.first << endl;

cout << "D" << (Round + 1) << ": " << halves.second << endl;

roundKeys[Round] = PC\_2(halves.first, halves.second);

cout << "Key" << Round + 1 << ": " << roundKeys[Round] << endl << endl;

}

return roundKeys;

}

string XOR(const string& a, const string& b)

{

string result;

for (size\_t i = 0; i < a.size(); i++)

{

result += to\_string((a[i] - '0') ^ (b[i] - '0'));

}

return result;

}

string expansionPermutation(const string& input)

{

string expanded;

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

{

expanded += input[E\_BIT\_SELECTION\_TABLE[i] - 1];

}

return expanded;

}

string sBoxSubstitution(const string& input)

{

string output;

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

{

int row = 2 \* (input[6 \* i] - '0') + (input[6 \* i + 5] - '0');

int col = 8 \* (input[6 \* i + 1] - '0') + 4 \* (input[6 \* i + 2] - '0') +

2 \* (input[6 \* i + 3] - '0') + (input[6 \* i + 4] - '0');

int val = S\_BOXES[i][row][col];

output += bitset<4>(val).to\_string();

}

return output;

}

string inverse\_initial\_permutation(const string& input)

{

string permuted;

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

{

permuted += input[INVERSE\_INITIAL\_PERMUTATION[i] - 1];

}

return permuted;

}

string Permutation(const string& input)

{

string permuted;

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

{

permuted += input[PERMUTATION\_TABLE[i] - 1];

}

return permuted;

}

string DESRound(const string& L, const string& R, const string& key)

{

string expandedR = expansionPermutation(R);

cout << "Expanded R: " << expandedR << endl;

string xored = XOR(expandedR, key);

cout << "XOR with particular key: " << xored << endl;

string sBoxOutput = sBoxSubstitution(xored);

cout << "S-Box Output: " << sBoxOutput << endl;

string permuted = Permutation(sBoxOutput);

cout << "Permuted: " << permuted << endl;

return XOR(L, permuted);

}

string binaryToAscii(const string& binary)

{

string ascii = "";

for (size\_t i = 0; i < binary.length(); i += 8)

{

string byte = binary.substr(i, 8);

char character = static\_cast<char>(bitset<8>(byte).to\_ulong());

ascii += character;

}

return ascii;

}

int main()

{

string key = "0011010000101101101101011010100000011101110110111001000000000100";

cout << "64 bit Key: " << key << endl;

pair<string, string> halves = PC\_1(key);

cout << "After PC-1 56 bit key: " << halves.first + halves.second << endl;

cout << endl;

vector<string> roundKeys = performRounds(halves);

string pt = "0100000101101110011100110110000101110010011110011000000010000000";

string iv = "0000000000000000000000000000000000000000000000000000000000000000";

string permuted\_pt = initial\_permutation(XOR(pt, iv));

cout << "After initial permutation: " << endl;

string L0 = permuted\_pt.substr(0, 32);

string R0 = permuted\_pt.substr(32, 32);

cout << "L0: " << L0 << endl;

cout << "R0: " << R0 << endl;

cout << endl;

string L = L0, R = R0;

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

cout << "Round " << i + 1 << " Encryption: " << endl;

string newL = R;

R = DESRound(L, R, roundKeys[i]);

L = newL;

cout << "After XOR" << ": L"<< i+1 << " = "<< L << ", R"<< i+1 << " = " << R << endl << endl;

}

string swapedLR = R + L;

cout << "Swapped value: " << swapedLR << endl;

string ciphertext = inverse\_initial\_permutation(swapedLR);

cout << "Ciphertext: " << ciphertext << endl;

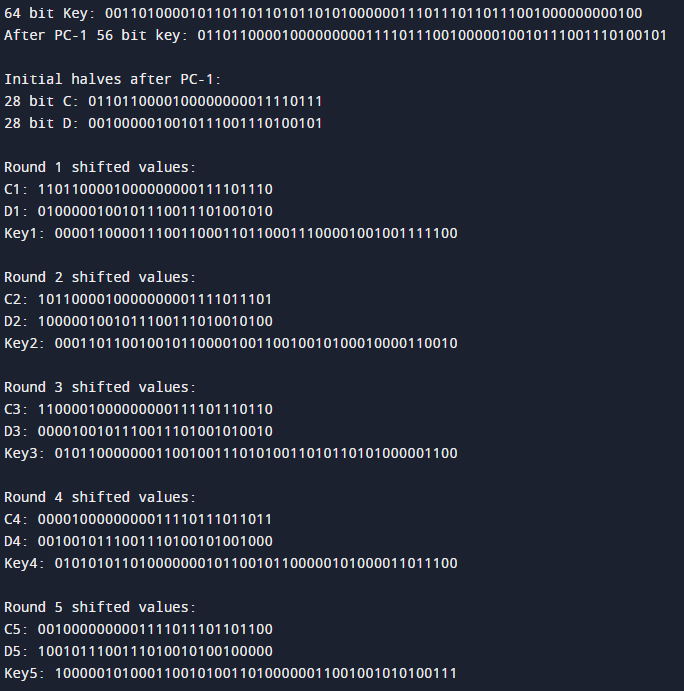
string asciiCiphertext = binaryToAscii(ciphertext);

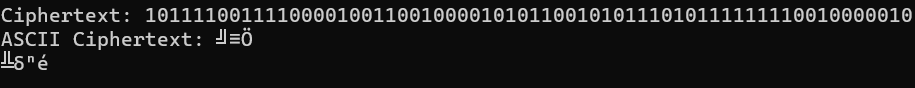
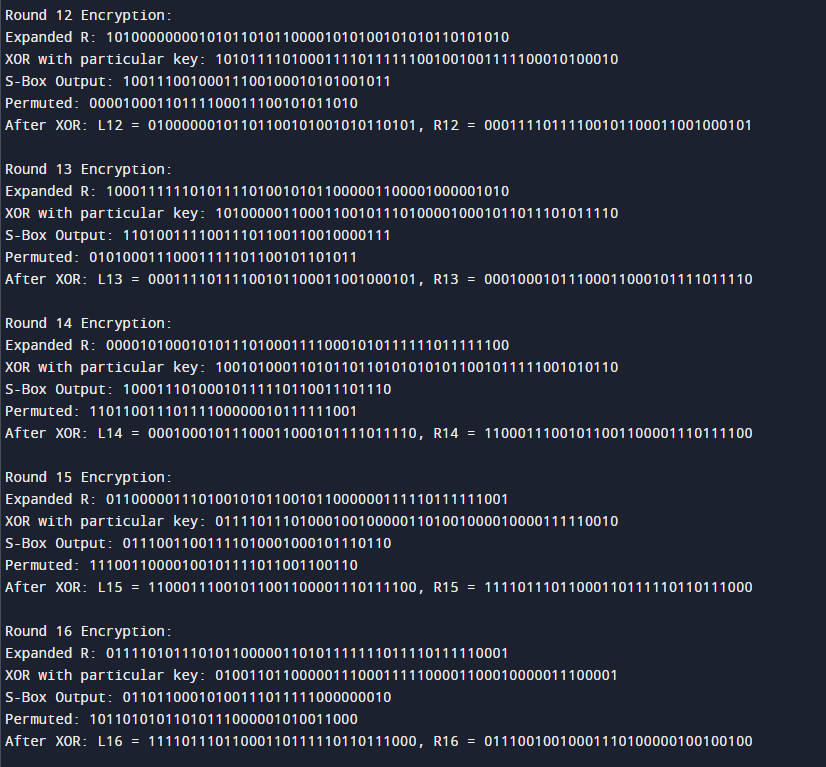
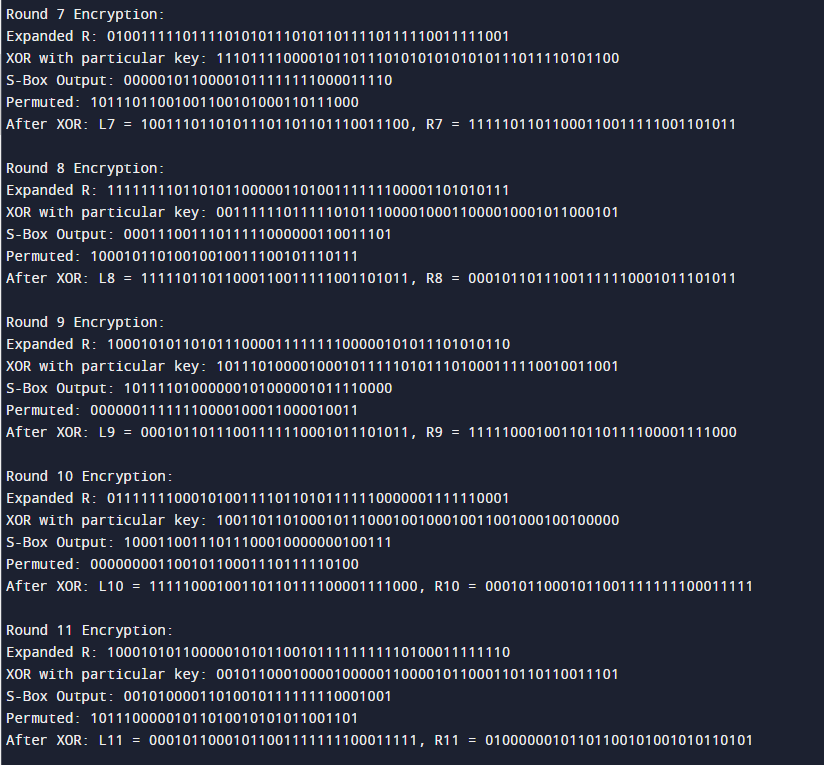
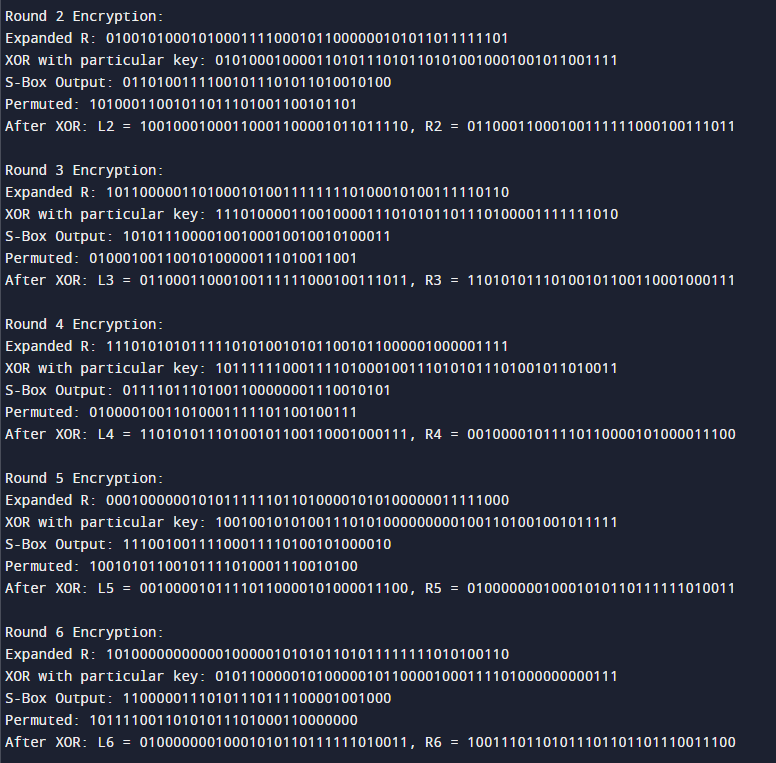
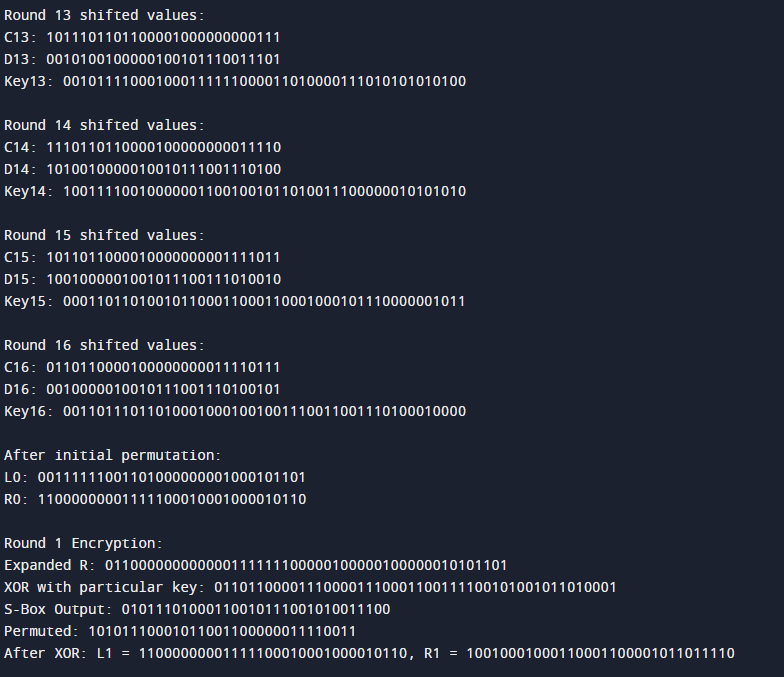
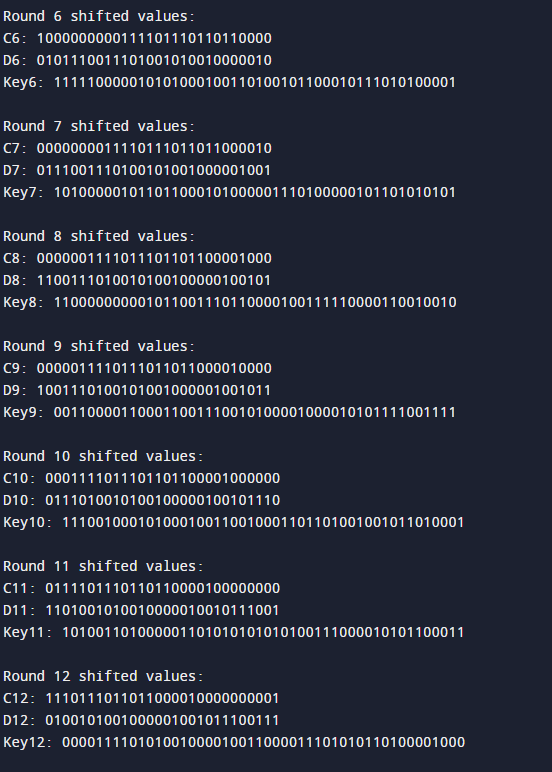
cout << "ASCII Ciphertext: " << asciiCiphertext << endl;

return 0;

}

**Output:**

****

****