A Mini Project Report on

DES 64-BIT

Submitted in partial fulfilment of the course

CSE-1007: Introduction to Cryptography

By

Group-15

20BCE7067	Madhyam Patra
20BCE7060	Vemula Venkat Rao
18BCE7276	Harshith Sai Teja Doda
18BCE7086	Narakula Sai Pavan Kumar
20BCE7236	Sudulagunta Brahmanya Ashrith

VIT®
AP

School of Computer Science & Engineering

VIT-AP UNIVERSITY

INAVOLU

DECEMBER, 2021

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

Up until recently, the main standard for encrypting data was a symmetric algorithm known as the Data Encryption Standard (DES). However, this has now been replaced by a new standard known as the Advanced Encryption Standard (AES) which we will look at later. DES is a 64 bit block cipher which means that it encrypts data 64 bits at a time. This is contrasted to a stream cipher in which only one bit at a time (or sometimes small groups of bits such as a byte) is encrypted. DES was the result of a research project set up by International Business Machines (IBM) corporation in the late 1960's which resulted in a cipher known as LUCIFER. In the early 1970's it was decided to commercialise LUCIFER and a number of significant changes were introduced. IBM was not the only one involved in these changes as they sought technical advice from the National Security Agency (NSA) (other outside consultants were involved but it is likely that the NSA were the major contributors from a technical point of view). The altered version of LUCIFER was put forward as a proposal for the new national encryption standard requested by the National Bureau of Standards (NBS)3 . It was finally adopted in 1977 as the Data Encryption Standard - DES (FIPS PUB 46). Some of the changes made to LUCIFER have been the subject of much controversy even to the present day. The most notable of these was the key size. LUCIFER used a key size of 128 bits however this was reduced to 56 bits for DES. Even though DES actually accepts a 64 bit key as input, the remaining eight bits are used for parity checking and have no effect on DES's security. Outsiders were convinced that the 56 bit key was an easy target for a brute force attack4 due to its extremely small size. The need for the parity checking scheme was also questioned without satisfying answers. Another controversial issue was that the S-boxes used were designed under classified conditions and no reasons for their particular design were ever given. This led people to assume that the

NSA had introduced a “trapdoor” through which they could decrypt any data encrypted by DES even without knowledge of the key.

One startling discovery was that the S-boxes appeared to be secure against an attack known as Differential Cryptanalysis which was only publicly discovered by Biham and Shamir in 1990. This suggests that the NSA were aware of this attack in 1977; 13 years earlier! In fact the DES designers claimed that the reason they never made the design specifications for the S-boxes available was that they knew about a number of attacks that weren’t public knowledge at the time and they didn’t want them leaking - this is quite a plausible claim as differential cryptanalysis has shown.

However, despite all this controversy, in 1994 NIST reaffirmed DES for government use for a further five years for use in areas other than “classified”. DES of course isn’t the only symmetric cipher. There are many others, each with varying levels of complexity. Such ciphers include: IDEA, RC4, RC5, RC6 and the new Advanced Encryption Standard (AES). AES is an important algorithm and was originally meant to replace DES (and its more secure variant triple DES) as the standard algorithm for non-classified material.

THEORETICAL STUDY

The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).

DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only). General Structure of DES is depicted in the following illustration.

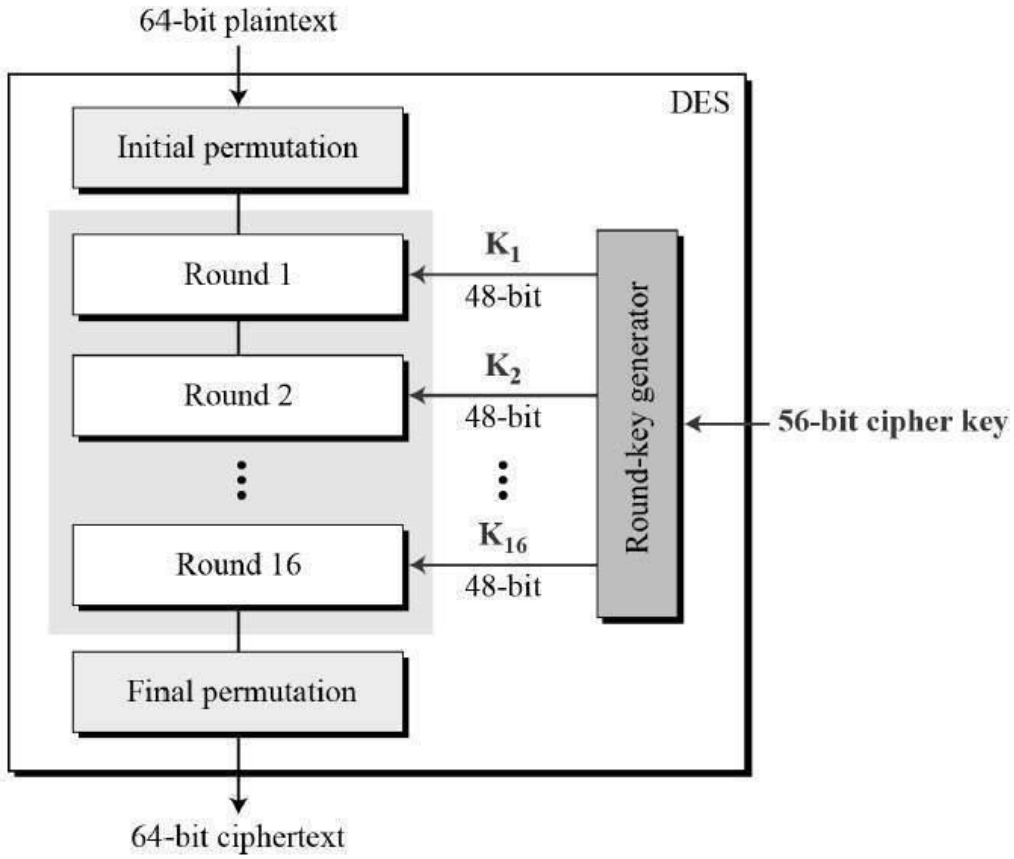


Fig 1:Working of DES

IMPLEMENTATION

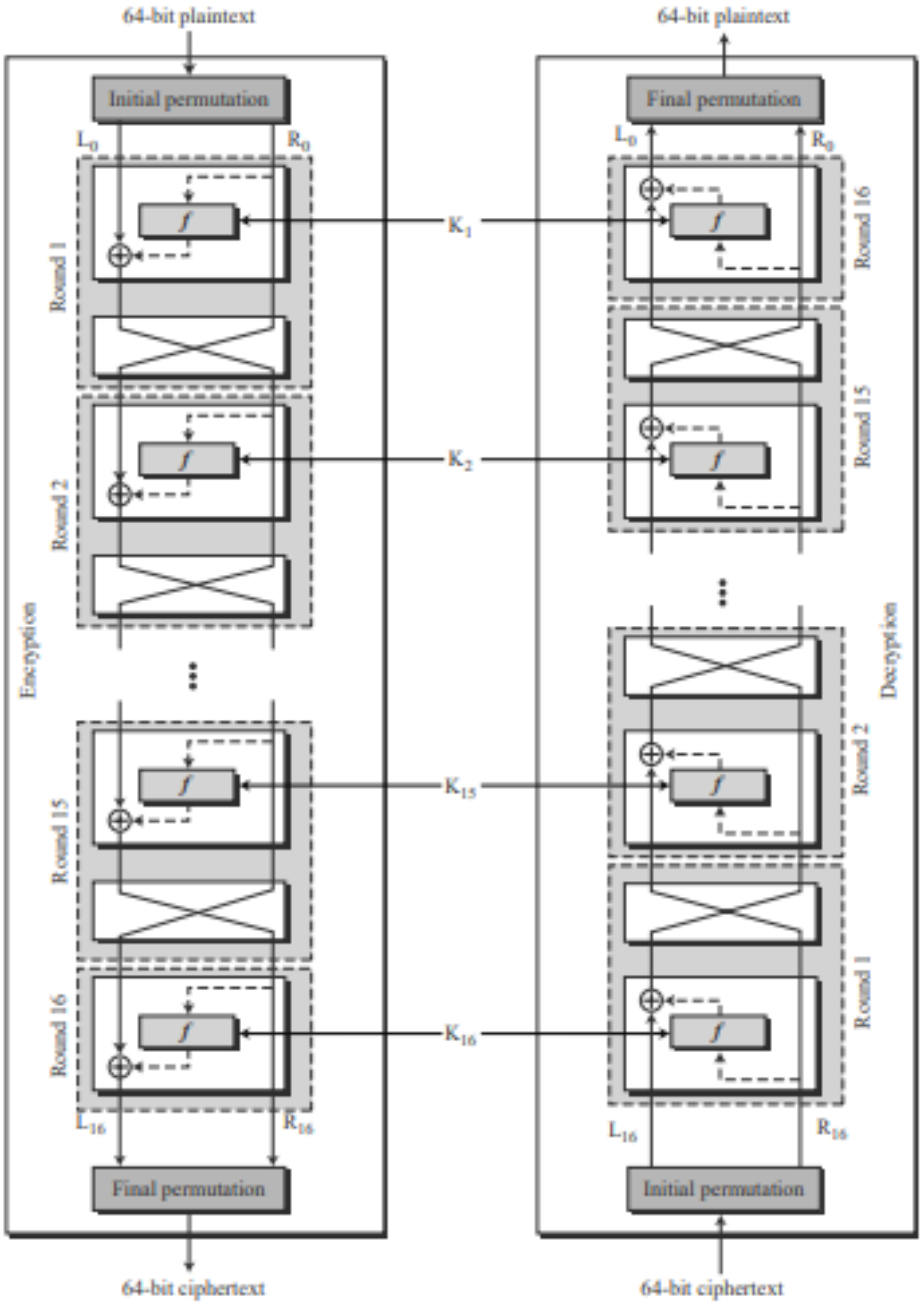


Fig 2: DES and Reverse DES Cipher

Since DES is based on the Feistel Cipher, all that is required to specify DES is

- Round function
- Key schedule
- Any additional processing − Initial and final permutation

INITIAL AND FINAL PERMUTATION

The initial and final permutations are straight Permutation boxes (P-boxes) that are inverses of each other. They have no cryptography significance in DES. The initial and final permutations are shown as follows

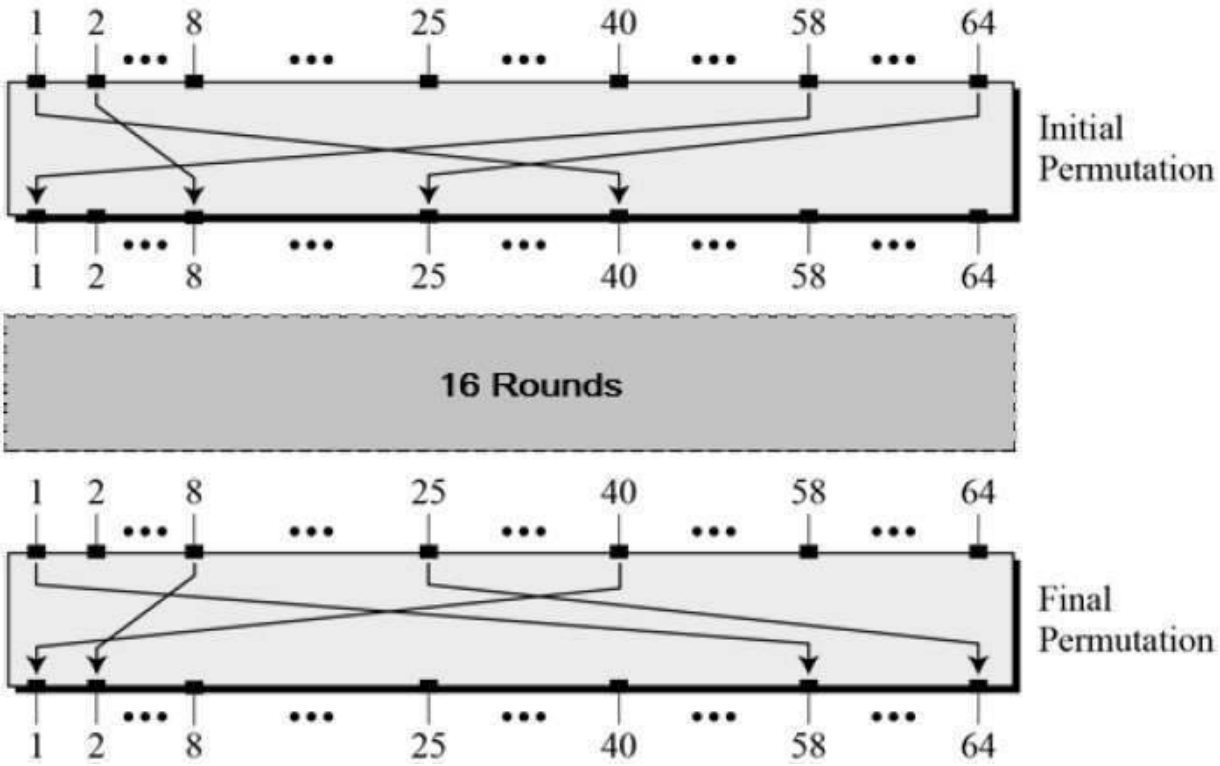
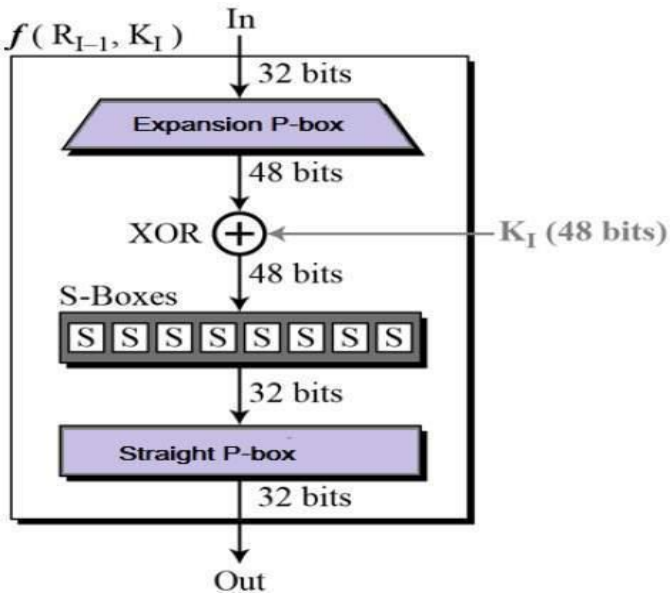


Fig 3:Initial and Final Permutation

DES ROUND FUNCTION

The heart of this cipher is the DES function, *f*. The DES function applies a



48-bit key to the rightmost 32 bits to produce a 32-bit output.

Fig 4:DES Round Function

Expansion Permutation Box

Since right input is 32-bit and round key is a 48-bit, we first need to expand right input to 48 bits. Permutation logic is graphically depicted in the following illustration

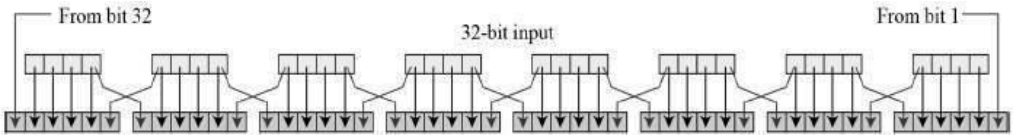


Fig 5:Expansion Permutation Box

- The graphically depicted permutation logic is generally described as table in DES specification illustrated as shown −

32	01	02	03	04	05
04	05	06	07	08	09
08	09	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	31	31	32	01

- **XOR (Whitener).** − After the expansion permutation, DES does XOR operation on the expanded right section and the round key. The round key is used only in this operation.

Substitution Boxes

The S-boxes carry out the real mixing (confusion). DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output. Refer the following illustration −

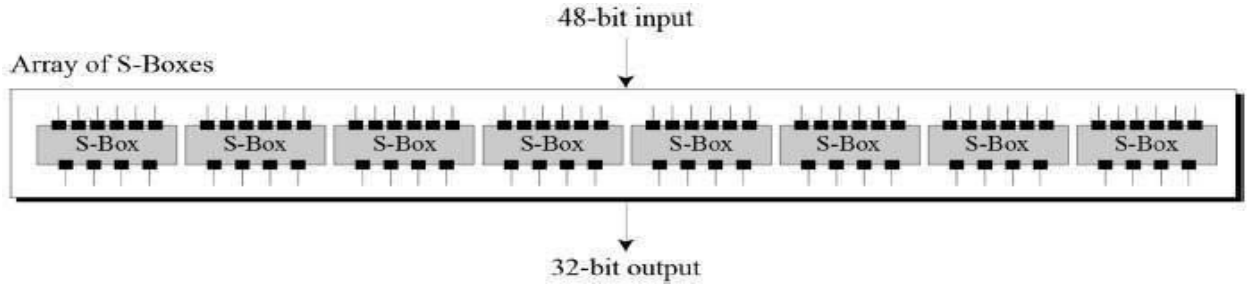
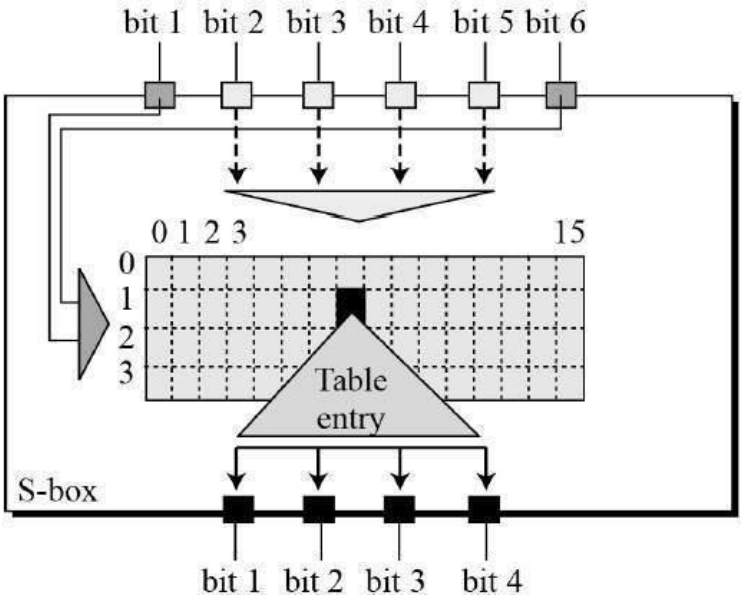


Fig 6:Substitution Boxes

- The S-box rule is illustrated below



There are a total of eight S-box tables. The output of all eight s-boxes is then combined in to 32 bit section.

Straight Permutation

The 32 bit output of S-boxes is then subjected to the straight permutation with rule shown in the following illustration:

16	07	20	21	29	12	28	17
01	15	23	26	05	18	31	10
02	08	24	14	32	27	03	09
19	13	30	06	22	11	04	25

KEY GENERATION

The round-key generator creates sixteen 48-bit keys out of a 56-bit cipher key. The process of key generation is depicted in the following illustration −

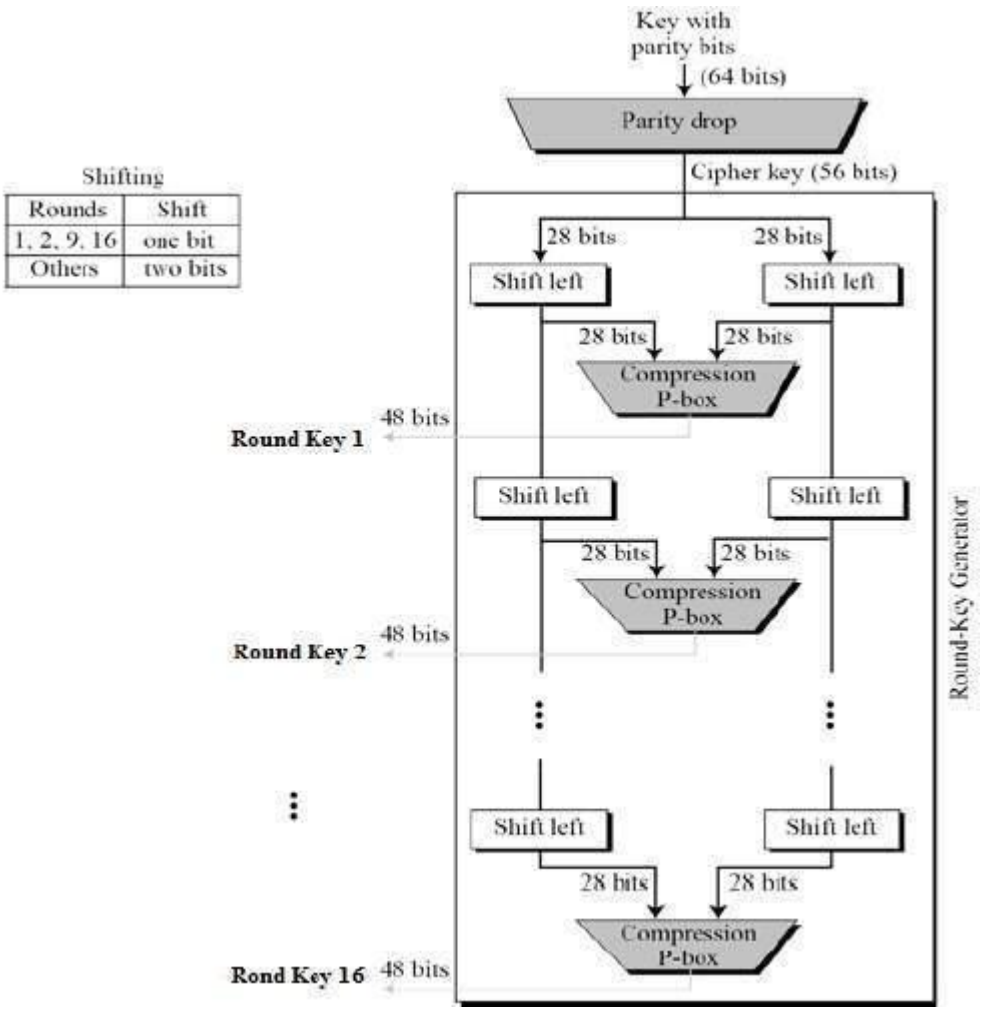


Fig 7:Key Generation

The logic for Parity drop, shifting, and Compression P-box is given in the DES description.

PSEUDO CODE/ALGORITHM

```
import java.util.*;

public class DES {
      private static final byte[] IP = {
                  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
            };

            // Permuted Choice 1 table
            private static final byte[] PC1 = {
                  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
            };

            // Permuted Choice 2 table
            private static final byte[] PC2 = {
                  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
            };
```

```
            // Array to store the number of rotations that
are to be done on each round
            private static final byte[] rotations = {
                  1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2,
2, 1
            };

            // Expansion (aka P-box) table
            private static final byte[] 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
            };

            // S-boxes (i.e. Substitution boxes)
            private static final byte[][] S = { {
                  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
            } };
            // Permutation table
            private static final byte[] 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
            };

            // Final permutation (aka Inverse permutation)
table
            private static final byte[] FP = {
                  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
            };

            // 28 bits each, used as storage in the KS
(Key Structure) rounds to
            // generate round keys (aka subkeys)
            private static int[] C = new int[28];
            private static int[] D = new int[28];

            // Decryption requires the 16 subkeys to be
used in the exact same process
            // as encryption, with the only difference
being that the keys are used
            // in reverse order, i.e. last key is used
first and so on. Hence, during
```

```
		// encryption when the keys are first
generated, they are stored in this
		// array. In case we wish to separate the
encryption and decryption
		// programs, then we need to generate the
subkeys first in order, store
		// them and then use them in reverse order.
		private static int[][] subkey = new
int[16][48];

		public static void main(String args[]) {
			System.out.println("Enter the input as a
16 character hexadecimal value:");
			String input = new
Scanner(System.in).nextLine();
			int inputBits[] = new int[64];
			// inputBits will store the 64 bits of
the input as a an int array of
			// size 64. This program uses int arrays
to store bits, for the sake
			// of simplicity. For efficient
programming, use long data type. But
			// it increases program complexity which
is unnecessary for this
			// context.
			for(int i=0 ; i < 16 ; i++) {
				// For every character in the 16
bit input, we get its binary value
				// by first parsing it into an int
and then converting to a binary
				// string
				String s =
Integer.toBinaryString(Integer.parseInt(input.charAt(i) +
"", 16));

				// Java does not add padding zeros,
i.e. 5 is returned as 111 but
				// we require 0111. Hence, this
while loop adds padding 0's to the
				// binary value.
				while(s.length() < 4) {
					s = "0" + s;
				}
```

```
                        // Add the 4 bits we have extracted
into the array of bits.
                        for(int j=0 ; j < 4 ; j++) {
                              inputBits[(4*i)+j] =
Integer.parseInt(s.charAt(j) + "");
                        }
                  }

                  // Similar process is followed for the 16
bit key
                  System.out.println("Enter the key as a 16
character hexadecimal value:");
                  String key = new
Scanner(System.in).nextLine();
                  int keyBits[] = new int[64];
                  for(int i=0 ; i < 16 ; i++) {
                        String s =
Integer.toBinaryString(Integer.parseInt(key.charAt(i) +
"", 16));
                        while(s.length() < 4) {
                              s = "0" + s;
                        }
                        for(int j=0 ; j < 4 ; j++) {
                              keyBits[(4*i)+j] =
Integer.parseInt(s.charAt(j) + "");
                        }
                  }

                  // permute(int[] inputBits, int[]
keyBits, boolean isDecrypt)
                  // method is used here. This allows
encryption and decryption to be
                  // done in the same method, reducing
code.
                  System.out.println("\n+++ ENCRYPTION
+++");
                  int outputBits[] = permute(inputBits,
keyBits, false);
                  System.out.println("\n+++ DECRYPTION
+++");
                  permute(outputBits, keyBits, true);
            }
```

```
          private static int[] permute(int[] inputBits,
int[] keyBits, boolean isDecrypt) {
                // Initial permutation step takes input
bits and permutes into the
                // newBits array
                int newBits[] = new
int[inputBits.length];
                for(int i=0 ; i < inputBits.length ; i++)
{
                      newBits[i] = inputBits[IP[i]-1];
                }

                // 16 rounds will start here
                // L and R arrays are created to store
the Left and Right halves of the
                // subkey respectively
                int L[] = new int[32];
                int R[] = new int[32];
                int i;

                // Permuted Choice 1 is done here
                for(i=0 ; i < 28 ; i++) {
                      C[i] = keyBits[PC1[i]-1];
                }
                for( ; i < 56 ; i++) {
                      D[i-28] = keyBits[PC1[i]-1];
                }

                // After PC1 the first L and R are ready
to be used and hence looping
                // can start once L and R are initialized

                System.arraycopy(newBits, 0, L, 0, 32);
                System.arraycopy(newBits, 32, R, 0, 32);
                System.out.print("\nL0 = ");
                displayBits(L);
                System.out.print("R0 = ");
                displayBits(R);
                for(int n=0 ; n < 16 ; n++) {
                      System.out.println("\n-------------
");
                      System.out.println("Round " + (n+1)
+ ":");
```

```
                        // newR is the new R half generated
by the Fiestel function. If it
                        // is encrpytion then the KS method
is called to generate the
                        // subkey otherwise the stored
subkeys are used in reverse order
                        // for decryption.
                        int newR[] = new int[0];
                        if(isDecrypt) {
                              newR = fiestel(R, subkey[15-
n]);
                              System.out.print("Round key =
");
                              displayBits(subkey[15-n]);
                        } else {
                              newR = fiestel(R, KS(n,
keyBits));
                              System.out.print("Round key =
");
                              displayBits(subkey[n]);
                        }
                        // xor-ing the L and new R gives
the new L value. new L is stored
                        // in R and new R is stored in L,
thus exchanging R and L for the
                        // next round.
                        int newL[] = xor(L, newR);
                        L = R;
                        R = newL;
                        System.out.print("L = ");
                        displayBits(L);
                        System.out.print("R = ");
                        displayBits(R);
                  }

                  // R and L has the two halves of the
output before applying the final
                  // permutation. This is called the
"Preoutput".

                  int output[] = new int[64];
                  System.arraycopy(R, 0, output, 0, 32);
                  System.arraycopy(L, 0, output, 32, 32);
                  int finalOutput[] = new int[64];
```

```
                // Applying FP table to the preoutput, we
get the final output:
                // Encryption => final output is
ciphertext
                // Decryption => final output is
plaintext
                for(i=0 ; i < 64 ; i++) {
                      finalOutput[i] = output[FP[i]-1];
                }

                // Since the final output is stored as an
int array of bits, we convert
                // it into a hex string:
                String hex = new String();
                for(i=0 ; i < 16 ; i++) {
                      String bin = new String();
                      for(int j=0 ; j < 4 ; j++) {
                            bin += finalOutput[(4*i)+j];
                      }
                      int decimal = Integer.parseInt(bin,
2);
                      hex +=
Integer.toHexString(decimal);
                }
                if(isDecrypt) {
                      System.out.print("Decrypted text:
");
                      } else {
                      System.out.print("Encrypted text:
");
                }
                System.out.println(hex.toUpperCase());
                return finalOutput;
          }

          private static int[] KS(int round, int[] key)
{
                // The KS (Key Structure) function
generates the round keys.
                // C1 and D1 are the new values of C and
D which will be generated in
                // this round.
                int C1[] = new int[28];
                int D1[] = new int[28];
```

```
                // The rotation array is used to set how
many rotations are to be done

                int rotationTimes = (int)
rotations[round];
                // leftShift() method is used for
rotation (the rotation is basically)
                // a left shift operation, hence the
name.
                C1 = leftShift(C, rotationTimes);
                D1 = leftShift(D, rotationTimes);
                // CnDn stores the combined C1 and D1
halves
                int CnDn[] = new int[56];
                System.arraycopy(C1, 0, CnDn, 0, 28);
                System.arraycopy(D1, 0, CnDn, 28, 28);
                // Kn stores the subkey, which is
generated by applying the PC2 table
                // to CnDn
                int Kn[] = new int[48];
                for(int i=0 ; i < Kn.length ; i++) {
                      Kn[i] = CnDn[PC2[i]-1];
                }

                // Now we store C1 and D1 in C and D
respectively, thus becoming the
                // old C and D for the next round. Subkey
is stored and returned.
                subkey[round] = Kn;
                C = C1;
                D = D1;
                return Kn;
           }

           private static int[] fiestel(int[] R, int[]
roundKey) {
                // Method to implement Fiestel function.
                // First the 32 bits of the R array are
expanded using E table.
                int expandedR[] = new int[48];
                for(int i=0 ; i < 48 ; i++) {
                      expandedR[i] = R[E[i]-1];
                }
```

```
                // We xor the expanded R and the
generated round key
                int temp[] = xor(expandedR, roundKey);
                // The S boxes are then applied to this
xor result and this is the
                // output of the Fiestel function.
                int output[] = sBlock(temp);
                return output;




          }

          private static int[] xor(int[] a, int[] b) {
                // Simple xor function on two int arrays
                int answer[] = new int[a.length];
                for(int i=0 ; i < a.length ; i++) {
                      answer[i] = a[i]^b[i];
                }
                return answer;
          }

          private static int[] sBlock(int[] bits) {
                // S-boxes are applied in this method.
                int output[] = new int[32];
                // We know that input will be of 32 bits,
hence we will loop 32/4 = 8
                // times (divided by 4 as we will take 4
bits of input at each
                // iteration).
                for(int i=0 ; i < 8 ; i++) {
                      // S-box requires a row and a
column, which is found from the
                      // input bits. The first and 6th
bit of the current iteration
                      // (i.e. bits 0 and 5) gives the
row bits.
                      int row[] = new int [2];
                      row[0] = bits[6*i];
                      row[1] = bits[(6*i)+5];
                      String sRow = row[0] + "" + row[1];
                      // Similarly column bits are found,
which are the 4 bits between
```

```
                        // the two row bits (i.e. bits
1,2,3,4)
                        int column[] = new int[4];
                        column[0] = bits[(6*i)+1];
                        column[1] = bits[(6*i)+2];
                        column[2] = bits[(6*i)+3];
                        column[3] = bits[(6*i)+4];
                        String sColumn = column[0] +""+
column[1] +""+ column[2] +""+ column[3];
                        // Converting binary into decimal
value, to be given into the
                        // array as input
                        int iRow = Integer.parseInt(sRow,
2);
                        int iColumn =
Integer.parseInt(sColumn, 2);
                        int x = S[i][(iRow*16) + iColumn];
                        // We get decimal value of the S-
box here, but we need to convert
                        // it into binary:
                        String s =
Integer.toBinaryString(x);
                        // Padding is required since Java
returns a decimal '5' as '111' in
                        // binary, when we require '0111'.
                        while(s.length() < 4) {
                              s = "0" + s;
                        }
                        // The binary bits are appended to
the output
                        for(int j=0 ; j < 4 ; j++) {
                              output[(i*4) + j] =
Integer.parseInt(s.charAt(j) + "");
                        }
                  }
                  // P table is applied to the output and
this is the final output of one
                  // S-box round:
                  int finalOutput[] = new int[32];
                  for(int i=0 ; i < 32 ; i++) {
                        finalOutput[i] = output[P[i]-1];
                  }
                  return finalOutput;
            }
```

```
	private static int[] leftShift(int[] bits, int
n) {
		// Left shifting takes place here, i.e.
each bit is rotated to the left
		// and the leftmost bit is stored at the
rightmost bit. This is a left
		// shift operation.
		int answer[] = new int[bits.length];
		System.arraycopy(bits, 0, answer, 0,
bits.length);
		for(int i=0 ; i < n ; i++) {
			int temp = answer[0];
			for(int j=0 ; j < bits.length-1 ;
j++) {
				answer[j] = answer[j+1];
			}
			answer[bits.length-1] = temp;
		}
		return answer;
	}

	private static void displayBits(int[] bits) {
		// Method to display int array bits as a
hexadecimal string.

		for(int i=0 ; i < bits.length ; i+=4) {
			String output = new String();
			for(int j=0 ; j < 4 ; j++) {
				output += bits[i+j];
			}

	System.out.print(Integer.toHexString(Integer.parseIn
t(output, 2)));
		}
		System.out.println();
	}

}
```

RESULTS

Enter the input as a 16 character hexadecimal value:

1234567890ABCDEF

Enter the key as a 16 character hexadecimal value:

16518ABCEDEBF19D

+++ ENCRYPTION +++

L0 = cc1fc6e0

R0 = f0aae8a5

Round 1:

Round key = ff15054f3e24

L = f0aae8a5

R = 52228b48

Round 2:

Round key = 5b1d6b08d2f3

L = 52228b48

R = 609b90e5

Round 3:

Round key = e9e0f9d7ec25

L = 609b90e5

R = 4df64738

Round 4:

Round key = 95cf8eaa0fd8

L = 4df64738

R = 2aebeb71

Round 5:

Round key = 723b9399f317

L = 2aebeb71

R = 20077c64

Round 6:

Round key = 3dbc657746a0

L = 20077c64

R = 81d2a579

Round 7:

Round key = c364ded8294f

L = 81d2a579

R = f0c686d0

Round 8:

Round key = 7cc7b4a6f29c

L = f0c686d0

R = 92f1b085

Round 9:

Round key = f7dd208b2ccb

L = 92f1b085

R = decfab27

Round 10:

Round key = caabe36ef315

L = decfab27

R = d34833cc

Round 11:

Round key = b9f61f3345ee

L = d34833cc

R = 5f1e8fa4

Round 12:

Round key = 6517cacc9983

L = 5f1e8fa4

R = 139b3801

Round 13:

Round key = 72d8f5c6667d

L = 139b3801

R = 3210c36c

Round 14:

Round key = 9de9527b9bc8

L = 3210c36c

R = 6919ce0a

Round 15:

Round key = 2667bf90d53b

L = 6919ce0a

R = b9a84b12

Round 16:

Round key = 2ef785bd5dcc

L = b9a84b12

R = 5e69afb2

Encrypted text: 9C4F44FCC3B558A5

+++ DECRYPTION +++

L0 = 5e69afb2

R0 = b9a84b12

Round 1:

Round key = 2ef785bd5dcc

L = b9a84b12

R = 6919ce0a

Round 2:

Round key = 2667bf90d53b

L = 6919ce0a

R = 3210c36c

Round 3:

Round key = 9de9527b9bc8

L = 3210c36c

R = 139b3801

Round 4:

Round key = 72d8f5c6667d

L = 139b3801

R = 5f1e8fa4

Round 5:

Round key = 6517cacc9983

L = 5f1e8fa4

R = d34833cc

Round 6:

Round key = b9f61f3345ee

L = d34833cc

R = decfab27

Round 7:

Round key = caabe36ef315

L = decfab27

R = 92f1b085

Round 8:

Round key = f7dd208b2ccb

L = 92f1b085

R = f0c686d0

Round 9:

Round key = 7cc7b4a6f29c

L = f0c686d0

R = 81d2a579

Round 10:

Round key = c364ded8294f

L = 81d2a579

R = 20077c64

Round 11:

Round key = 3dbc657746a0

L = 20077c64

R = 2aebeb71

Round 12:

Round key = 723b9399f317

L = 2aebeb71

R = 4df64738

Round 13:

Round key = 95cf8eaa0fd8

L = 4df64738

R = 609b90e5

Round 14:

Round key = e9e0f9d7ec25

L = 609b90e5

R = 52228b48

Round 15:

Round key = 5b1d6b08d2f3

L = 52228b48

R = f0aae8a5

Round 16:

Round key = ff15054f3e24

L = f0aae8a5

R = cc1fc6e0

Decrypted text: 1234567890ABCDEF

REFERENCES

[1] "**DES Algorithm" Illustrated by J. Orlin Grabbe**

http://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm

[2] Sanjoy Kumer Deb Jun. 11, 19 · "DES" Security Zone · Presentation

https://dzone.com/articles/security-algorithms-des-algorithm

[3] IITKGP Sourav "THE DATA ENCRYPTION STANDARD"

http://www.facweb.iitkgp.ac.in/~sourav/DES.pdf