Data Encryption Standard (DES) Algorithm

Yang Liuxin

UCD Student Number: *19206207*
BJUT Student Number: *19372226*

Contents

1 Introduction

This document explains the implementation of the Data encryption standard (DES) Algorithm. The following table shows my technology stack. The following shows the requirements that are met in my implementation.

Programming Language	Java
Tool	IntelliJ IDEA

- implement both the encryption and the decryption parts of the DES algorithm without placing any restriction on the length of the plain/cipher text or the length of the key;
- design and implement a user-friendly interface in which the plain/cipher text can be drawn from the file system or input through the interface;
- make encryption and decryption two independent components in that a key needs to be supplied when performing either component.

Now I will explore these in detail.

2 Des Algorithm

2.1 Encryption

DES is a block cipher that encrypts data in 64-bit blocks. This implies that plaintext is divided into blocks, which act as the input to DES and then generate the ciphertext. I encrypt each block separately and later concatenate them together. Encryption and decryption employ the same algorithm and key, with slight variations. The key is 56 bits long. The concept is shown in Figure 1:

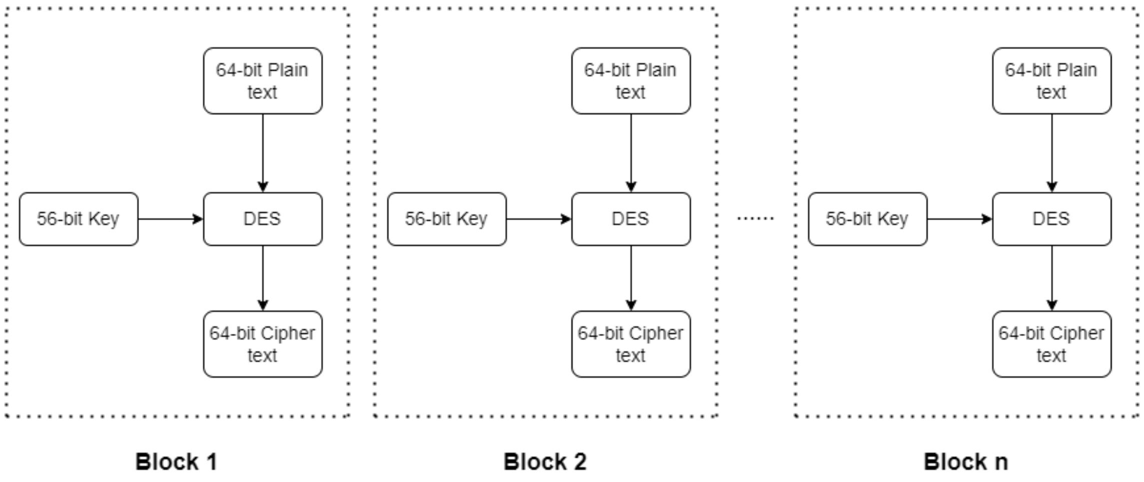
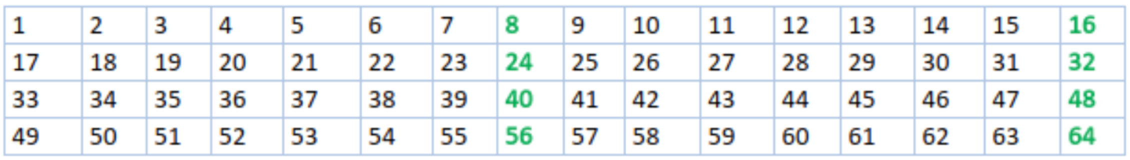


Figure 1: Des Algorithm Overview

Although the DES algorithm uses a key of 56 bits, the initial key is actually 64-bits long. This is because the bits at the positions that are multiples of 8, ie, 8, 16, 24, 32, 40, 48, 56 and 64, are removed from the key composition (see Figure 2). Therefore, the initial key of 64 bits long is converted into a 56-bit key.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Figure 2: Discard every 8_th bit of the initial key

The following shows the main steps of DES algorithm (see Figure 3 and Figure 4).

1. Perform the initial permutation (IP) on the initial 64-bit plain text.

2. Two halves, which are referred to as Left Plain Text (LPT) and Right Plain Text (RPT), is generated after performing the initial permutation.
3. Perform the 16 rounds of encryption on the Left Plain Text and Right Plain Text. In each round, the algorithm performs the substitution and transposition process.
4. Concatenate the Left Plain Text and Right Plain Text, which are put into the final/inverse permutation (FP).
5. Now I have the ciphertext of 64 bits long.

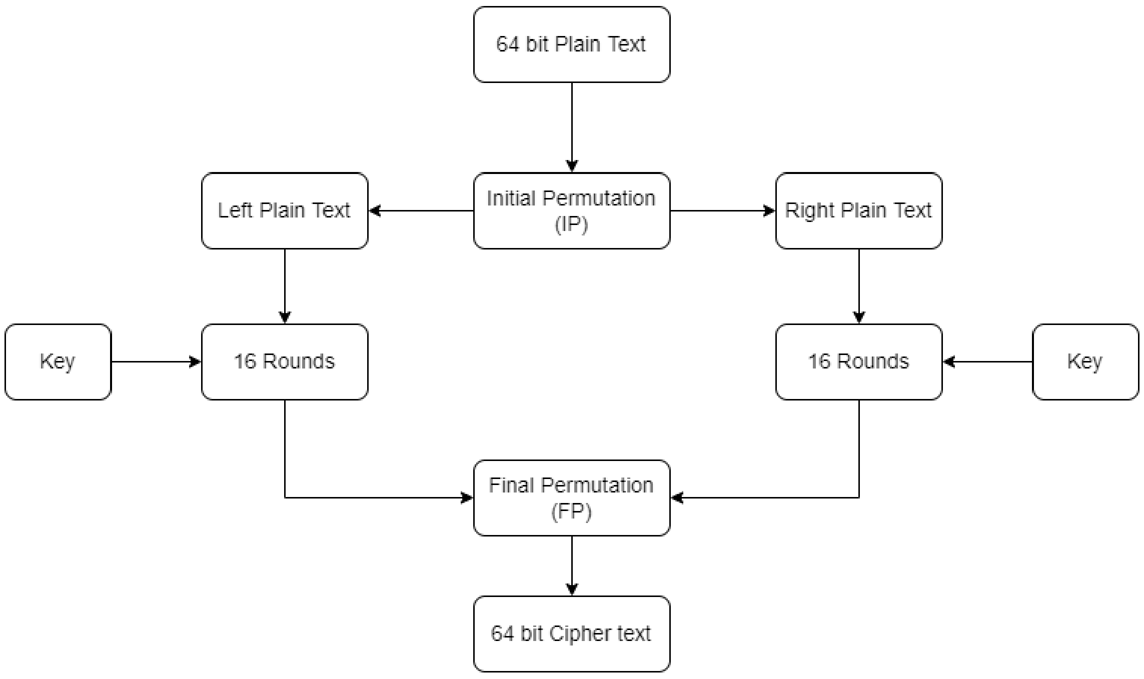


Figure 3: Steps of DES

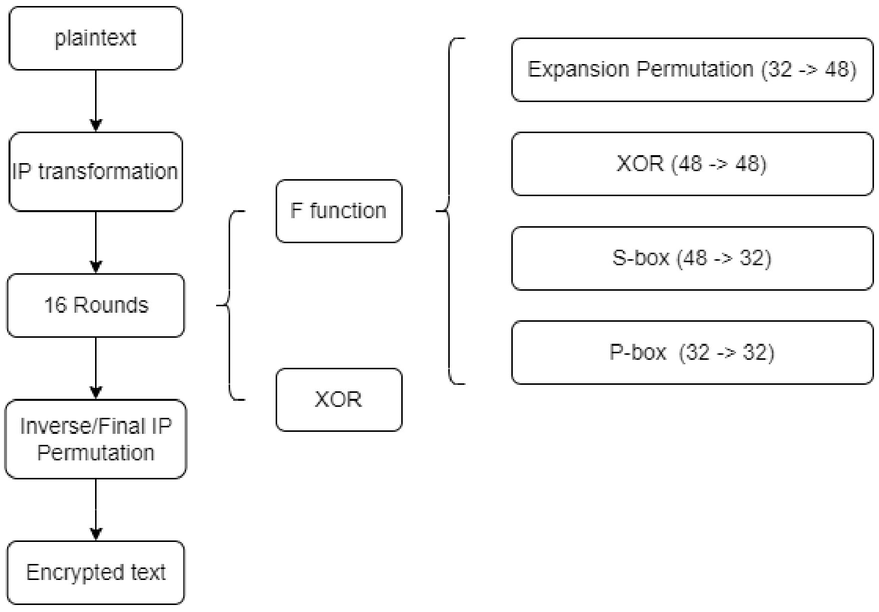


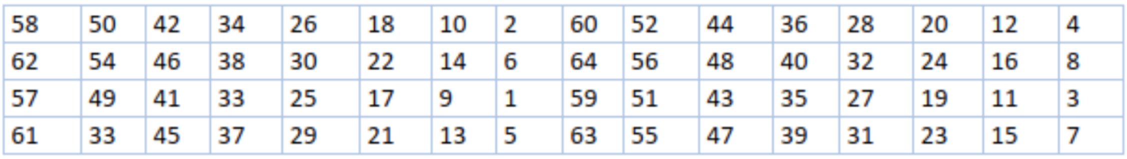
Figure 4: Steps of DES

The detailed codes are shown as follows.

```
    // round number of encryption
    private static final int ENCRY_COUNT = 16;
    // round number of decryption
    private static final int DECIP_COUNT = 15;
```

```java
    // flag: 1 represents encryption; 0 represents decryption
    public byte[] Encrypt(int[] encrypt_data, int flag, int[][] key_array) {
        int i;
        // Initial permutation of plaintext by initial permutation function
        for (i = 0; i < 64; i++) {
            // Plaintext IP conversion
            M[i] = encrypt_data[INIT_REP_IP[i] - 1];
        }
        // encryption
        if (flag == 1) {
            for (i = 0; i < ENCRY_COUNT; i++) {
                M = LoopF(M, i, flag, key_array);
            }
        }
        // decryption
        else if (flag == 0) {
            for (i = DECIP_COUNT; i >= 0; i--) {
                M = LoopF(M, i, flag, key_array);
            }
        }
        // Perform the inverse IP_1 operation
        for (i = 0; i < 64; i++) {
            MIP_1[i] = M[INIT_INVER_REP_IP[i] - 1];
        }
        // Return encrypted data
        encrypt = Util.binaryIntArrayToInt(MIP_1);
        return encrypt;
    }
```

Firstly, I perform the **initial permutation** process (see Figure 3). The initial permutation process happens one time before the first round. The first bit of the original plain text block is substituted with the 58th bit, and similar rules are shown in Figure 5.



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	33	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Figure 5: Initial Permutation Table

Having finished the initial permutation process, I have two halves, i.e., Left Plain Text (LPT) and Right Plain Text (RPT). Each of these two blocks is composed of 32 bits, which I will then apply the 16 rounds of key transformation, expansion permutation, S-box permutation, P-box permutation and XOR and Swap processes in turn.

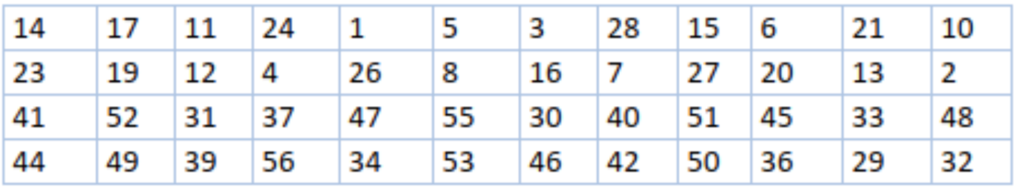
1. **Key transformation**

   Since I have discarded the bits in positions that are multiples of 8, I now have a key of 56 bits long, from which various sub key of 48 bits long is produced in each round. This process is known as the key transformation.

   48 bits are selected from 56 bits using the following Table 6. This process is known as compression permutation. Since different subsets of key bits are employed according to the compression permutation, DES algorithm is relatively difficult to crack.

2. **Expansion permutation**

   After performing the initial permutation, we have derived two blocks, Left Plain Text and Right Plain Text of 32 bits. These 32 bits are divided into 8 blocks, each of which has 4 bit. Now I perform the expansion permutation so that each block grows to contain 6 bits (see Figure 7).



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

Figure 6: Compression Permutation

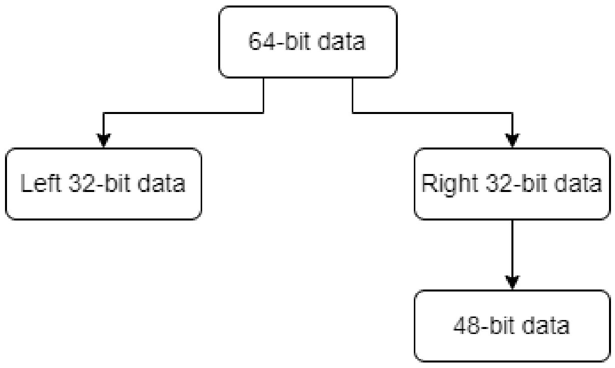


Figure 7: Expansion Permutation

3. **XOR**

   Recall that I have a 48-bit key compressed from 56-bit using the key transformation and applied the expansion permutation to expand the 32-bit to 48-bit. Therefore, keys and input data can be worked in the XOR statement.

4. **S-Box substitution**

   The S-Box Substitution uses the substitution approach to produce a 32-bit output from a 48-bit input generated from the XOR operation that contains the compressed key and expanded RPT. The S-boxes carry out the substitution. Each 8-S-box provides a 6-bit input and a 4-bit output. Each of the 8 sub-blocks (each with 6 bits) that make up the 48-bit input block is sent to an S-box. Each box's substitution follows a predetermined rule based on a 4-row by 16-column table. The input's bits one through six correspond to four rows, whereas bits two through five correspond to sixteen columns. Since each S-box has its own table, eight tables are needed to capture these boxes' output. An example is shown in Figure 8.

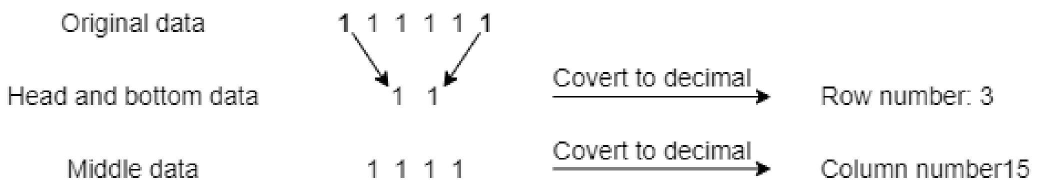


Figure 8: S-Box substitution Example

The corresponding codes are shown as follows.

```
    // compressed by S-box to 32-bit sValue
    for (i = 0; i < 8; i++) {
        for (j = 0; j < 6; j++) {
            // 48 is divided into 8 groups, 6 in each group
            S[i][j] = RE[(i * 6) + j];
        }
        // The following goes through the S-box to get a decimal number
        sBoxData[i] = S_Box[i][(S[i][0] << 1) + S[i][5]][(S[i][1] << 3) + (S
            [i][2] << 2) + (S[i][3] << 1) + S[i][4]];
        // The decimal number obtained in the S-box becomes 4-bit binary
        for (j = 0; j < 4; j++) {
            sValue[((i * 4) + 3) - j] = sBoxData[i] % 2;
```

```
                sBoxData[i] = sBoxData[i] / 2;
            }
        }
```

5. **P-box permutation**

   Similarly, replace the data with the data specified in the position of the p-box permutation. The corresponding codes are shown as follows.

```
    // sValue is transformed by P into 32-bit RP
    for (i = 0; i < 32; i++) {
        // P transformation
        RP[i] = sValue[P[i] - 1];
        // Move right to left
        L1[i] = R0[i];
        // The sum of L0 and RP is added bitwise (heterogeneous or operation
            ) to obtain R1
        R1[i] = L0[i] + RP[i];
        if (R1[i] == 2) {
            R1[i] = 0;
        }
        // Re-synthesize M and return the array M
        // In the last transformation, the left and right are not swapped.
            Here two transformations are used to achieve invariance
        if (((flag == 0) && (times == 0)) || ((flag == 1) && (times == 15)))
             {
            M[i] = R1[i];
            M[i + 32] = L1[i];
        } else {
            M[i] = L1[i];
               M[i + 32] = R1[i];
        }
    }
```

6. **Inverse Permutation**

   This process is similar to the initial permutation but with different tables.

2.2 Decryption

The decryption process is similar to the reverse of the encryption process that we have discussed in the last section.

3 More about Key Generation

Since my implementation allows the key to be any length, but I should use 8 byte of key, there is some design when I generate the key.

- If the key byte is less than 8 byte, I supply 0 to the empty space in the rest of 8 byte.
- If the key byte is more than 8 bytes, I supply 0 to the empty space in the rest of 16 bytes.

4 Functions

4.1 Encryption

- **Encrypt text**

  Firstly, convert key and input text into byte arrays, des_data and byte_key. Then use such byte arrays to encrypt the text. Then encrypt the data block by block (see the following codes).

```
    /**
     * des encryption
     *
     * @param des_data: data to be encrypted
     * @return the encrypted data after the encryption
     *
     */
    public byte[] desEncrypt(byte[] des_data, byte[] byte_key) {
        // convert into the standard data if expansion is needed
        byte[] format_key = DataFormat(byte_key);
        byte[] format_data = DataFormat(des_data);

        // length of the data
        int data_len = format_data.length;
        // construct a bye array to contain the encrypted data
        byte[] result_data = new byte[data_len];

        // Encryption
        // get the encrypted data
        // flag 1 represents encryption
        result_data = EcbModel(1, format_key, format_data, data_len,
            result_data);

        return result_data;
    }

    /**
     * Encryption/decryption of plaintext using ECB mode
     *
     * @param flag: 1 represents encryption and 0 represents decryption
     * @param format_key
     * @param format_data
     * @param data_len
     * @param result_data
     */
    public byte[] EcbModel(int flag, byte[] format_key, byte[] format_data,
        int data_len, byte[] result_data) {
        // Use the ECB mode ECB
        // encrypt 8 bytes each time
        int unit_count = data_len / 8;
        byte[] tmp_key = new byte[8];
        byte[] tmp_data = new byte[8];
        // Take the first eight bytes of the secret key after the formatting
            session
        System.arraycopy(format_key, 0, tmp_key, 0, 8);
        for (int i = 0; i < unit_count; i++) {
            // Take 8 bytes of the formatted data at a time
            System.arraycopy(format_data, i * 8, tmp_data, 0, 8);
            byte[] tmpresult = UnitDes(tmp_key, tmp_data, flag);
            System.arraycopy(tmpresult, 0, result_data, i * 8, 8);
        }
        return result_data;
    }
```

- **Encrypt text file**

  Firstly, read data stored in the text file. Consider it as text to perform encryption, which is similar to the process of encrypting text. After encryption, store the encrypted data in the specified location.

```
    // encrypt file
    public String encryptFile(String filepath, String key) {
        // convert the key into a byte array
        byte[] byte_key = key.getBytes();
        String f1 = System.getProperty("user.dir");
```

```java
        // get the absolute address of the directory that will store the
            encrypted and decrypted file
        String address = f1 + "\\src\\com\\assignment\\file\\plainText";
        File file = new File(filepath);
        if (!file.isDirectory()) {
            byte[] fileByte = Util.getFileByte(filepath);
            // encryption
            byte[] result = desEncrypt(fileByte, byte_key);
            String encryptedFileName = "encrypt_" + file.getName();
            // convert the encrypted byte array into String of base64 type
            String base64 = Base64.getEncoder().encodeToString(result);
            // write the base64 String into the file
            Util.writeIntoFile(address + "\\" + encryptedFileName, base64);
            System.out.println(base64);
            // return the absolute address where the encrypted file is
                stored
            return address + "\\" + encryptedFileName;
        }
        return "Sorry that directory encryption is not supported.";
    }
```

4.2 Decryption

- **Decrypt text**

```java
    /**
     * des decryption
     *
     * @param des_data: data to be decrypted
     * @return the decrypted data after the edcryption
     *
     */
    public byte[] desDecrypt(byte[] des_data, byte[] byte_key) {
        // convert into the standard data if expansion is needed
        byte[] format_key = DataFormat(byte_key);
        byte[] format_data = DataFormat(des_data);

        // length of the data
        int data_len = format_data.length;
        // construct a bye array to contain the encrypted data
        byte[] result_data = new byte[data_len];

        // Decryption
        // the purpose if to remove the padding bits generated during
            encryption
        // flag 0 represents decryption
        result_data = EcbModel(0, format_key, format_data, data_len,
            result_data);

        byte[] decrypt_byte_array = null;
        int delete_len = result_data[data_len - 8 - 1];
        delete_len = ((delete_len >= 1) && (delete_len <= 8)) ? delete_len :
             0;
        decrypt_byte_array = new byte[data_len - delete_len - 8];
        System.arraycopy(result_data, 0, decrypt_byte_array, 0, data_len -
            delete_len - 8);

        return decrypt_byte_array;
    }
```

- **Decrypt file**

  Similarly, read data from the encrypted file and use the key to decrypt it similar to text.

```java
    // decrypt file
    public String decryptFile(String filepath, String key) {
        // convert the key into a byte array
        byte[] bytekey = key.getBytes();
        String f1 = System.getProperty("user.dir");
        // get the absolute address of the directory that will store the encrypted and decrypted file
        String address = f1 + "\\src\\com\\assignment\\file\\plainText";
        // construct a file object
        File file = new File(filepath);
        // read the String base64 from file
        String encryptedBase64 = Util.readFromFile(file);
        System.out.println("encryptedBase64! " + encryptedBase64);
        // convert the base64 into byte array format
        byte[] result = Base64.getDecoder().decode(encryptedBase64);
        // decryption
        byte[] tem_result = desDecrypt(result, bytekey);
        String decryptedFileName = "decrypt_" + file.getName();
        Util.generateFile(tem_result, file.getParentFile().getAbsolutePath(), decryptedFileName);
        // return the absolute address where the decrypted file is stored
        return address + "\\" + decryptedFileName;
    }
```