```
Cryptologic and Security Implementations
Roll: Crs2115
DES Implementation
```

1 DES

DES is a block cipher. The main algorithm was developed by cryptographers at IBM. With several modifications made by the NSA. It was designed to withstand differential cryptanalysis. Despite some concerns about a “backdoor” installed by NSA (which could be used to break encryption), the modified algorithm was published in 1977 as the Data Encryption Standard.

DES encrypts 64 bit blocks using a 64 bit key (8 of whose bits are parity check bits). Only 56 bits completely define each key.

```
64 bit input
     |
     v
+-----------+
|           |
|    DES    |<--- 56 bit key
|           |
+-----------+
     |
     v
64 bit output
```

1.1 Structure

The input 64 bits is passed through an initial permutation IP. Then divided into 2 32 bit halves and passed through 16 rounds of a fiestel structure. Then through an inverse of the initial permutation.

Feistel Structure:

```
    |                             |
    v                             v
+---------------------------------------+
|     L             |           R       |
+---------------------------------------+
    |                             |
    |               +-------------|--------- round key
    |               |             |
    |               v             |
    |         +---------+         |
    -         |         |         |
   |+|<-------|    F    |<--------+
    -         |         |         |
    |         +---------+         |
    \                             /
     \---------------------------'
     /\
    |  \--------------------------+
    |                             |
    |                             |
    v                             v
+--------------------------------------+
|     L'           |           R'      |
+--------------------------------------+
    |                             |
    v                             v
```

The function F takes half of the input and a round key and generates an output which is used to mask the left half by xoring.

F is defined as follows:

- It takes the 32 bit half-block and expands it using an expansion matrix to 48 bits
- XORs the expanded block with the 48 bit round key
- Divides the 48 bit xored block into 8 6-bit blocks and uses each block to lookup 8 different S-boxes.
- Each S-box outputs 4 bits which are concateneted, and passed through a permutation P to create the 32 bit output.

During decryption most of the structure remains the same except that the feistel network becomes as follows:

```
     |                                 |
     v                                 v
+--------------------------------------------+
|      L                |            R        |
+--------------------------------------------+
     |                                 |
     |                 +---------------|--------- round key
     |                 |               |
     |                 v               |
     |           +---------+           |
     |           |         |           -
     |---------->|    F    |--------->|+|
     |           |         |           -
     |           +---------+           |
     \                                 /
      \-------------------------------'
      /\
     |  \------------------------------+
     |                                 |
     |                                 |
     v                                 v
+-------------------------------------------+
|      L'              |           R'      |
+-------------------------------------------+
     |                                 |
     v                                 v
```

1.2 Key Scheduling

The 56 bit main key is used to generate 16 48-bit round keys using a key scheduling algorithm.
The key scheduling algorithm can be described as follows:

- The 64 bit key(including parity bits) is passed through an “permutation-choice” PC-1 to create a 56 bit block.
- The following is repeated 16 times (16 rounds):
  - The 56 bit block is divided into 2 28 bit halves.
  - Each 28 bit half is rotated left, 1 or 2 times depending on the round r.
  - The 56 bit block is passed through another “permutation-choice” PC-2 to output the 48 bit round key k_r.

During decryption the key-schedule should provide the round keys in the reverse order.

1.3 OFB

The DES block cipher only encrypts 64 bit blocks. Several modes to a number of block cipher encryption blocks exist to encrypt arbitrary length plaintexts.
OFB turns essentialy the cipher into a stream cipher where the masking key is being generated by encrypting a random IV and the subsequent ciphertexts using the block cipher.

OFB (encryption):

```
+------------+
|     IV     |-----------.        .------------.        .------------.        .-  ...
+------------+           |        |            |        |            |        |
      |                  v        |            v        |            v        |
      |           +-------------+ |     +-------------+ |     +-------------+ |
      |           |             | |     |             | |     |             | |
      |           |     DES     | |     |     DES     | |     |     DES     | |
      |           |             | |     |             | |     |             | |
      |           +-------------+ |     +-------------+ |     +-------------+ |
      |                  |        |            |        |            |        |
      |                  |        |            |        |            |        |
      |                  |--------'            |--------'            |--------'
      |                  -                     -                     -
      |        .------->|+|          .------->|+|          .------->|+|
      |        |         -           |         -           |         -
      |        |         |           |         |           |         |
      v       m0         |          m1         |          m2         |
                         v                     v                     v
                        c0                    c1                    c2
```

During decryption the ciphertexts would be xored with the block cipher outputs instead of the plaintexts.

```
+------------+
|     IV     |-----------.        .------------.        .------------.        .-  ...
+------------+           |        |            |        |            |        |
      |                  v        |            v        |            v        |
      |           +-------------+ |     +-------------+ |     +-------------+ |
      |           |             | |     |             | |     |             | |
      |           |     DES     | |     |     DES     | |     |     DES     | |
      |           |             | |     |             | |     |             | |
      |           +-------------+ |     +-------------+ |     +-------------+ |
      |                  |        |            |        |            |        |
      |                  |        |            |        |            |        |
      |                  |--------'            |--------'            |--------'
      |                  -                     -                     -
      |        .------->|+|          .------->|+|          .------->|+|
      |        |         -           |         -           |         -
      |        |         |           |         |           |         |
      v       c0         |          c1         |          c2         |
                         v                     v                     v
                        m0                    m1                    m2
```

2 Implementation

2.1 Tools

The following functions are used to get or set the *idx*'th bit starting at an address *arr*, for any arbitrary *idx*.

```
int get_bit(unsigned char* arr, int idx)
{
    int x = idx/8;
    int y = idx%8;
    int ans = arr[x] & (1 << (7 - y));
    ans = !!ans; // clamp sets non-zero to 1, 0 to 0
    return ans;
}
void set_bit(unsigned char* arr, int idx, int bit)
{
    int x = idx/8;
    int y = idx%8;
    arr[x] = arr[x] | (bit << (7 - y));
}
```

2.2 Initial Permutation

The intial permutation can be performed using the permutation table and the previously defined functions as follows:

```
int IP[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 };

// initial permutation
void initial_permutation(unsigned char* block)
{
    unsigned char tmp_block[8];
    zero(tmp_block, 8);

    for(int i=0; i<64; i++)
        set_bit(tmp_block, i, get_bit(block, IP[i]-1));
        // tmp_block[i] = block[ IP[i]-1 ]
        // (logically tmp_block[i] accesses tmp_block's ith bit)

    for(int i=0; i<8; i++)
        block[i] = tmp_block[i];
}
```

The final permutation IP inverse is defined in a very similar manner.

2.3 S-Box lookup

S-Box implemented as folllows:

```
//  returns Sbox_idx[idx'th 6-bit block in ``block'']
int sbox_lookup(unsigned char* block, int idx)
{
    int i = idx*6; // bit number
    int row, col;

    row = (get_bit(block, i) << 1) | get_bit(block, i+5);

    col = get_bit(block, i+1);
    col = (col << 1) | get_bit(block, i+2);
    col = (col << 1) | get_bit(block, i+3);
    col = (col << 1) | get_bit(block, i+4);

    int **sbox = malloc(4 * sizeof(int*));
    for(int i=0; i<4; i++) sbox[i] = malloc(16*sizeof(int));

    sbox_selector(sbox, idx); // points double pointer sbox to idx'th sbox

    int ans = sbox[row][col];

    for(int i=0; i<4; i++) free(sbox[i]);
    free(sbox);

    return ans;
}
```

2.4 Function F

The function F is defined as follows:

```
void F(unsigned char* in, unsigned char* out, int round)
{
    unsigned char tmp[4];

    unsigned char expanded[6];
    unsigned char xored[6];
    zero(expanded, 6);
    zero(xored, 6);
    zero(tmp, 4);
    zero(out, 4);

    for(int i=0; i<48; i++)
        set_bit(expanded, i, get_bit(in, EXPANSION[i] - 1));
        // expanded[i] = in[EXPANSION[i] - 1]

    // xor with round_key
    for(int i=0; i<6; i++) xored[i] = expanded[i] ^ round_keys[round][i];

    for(int i=0; i<8; i++)
    {
        // sbox of ith 6 bit block
        int x = sbox_lookup(xored, i); // returns sbox_i[ith 6-bit block in xored]
        put_sub_block(tmp, i, 4, x); // puts value of x as i'th 4-bit block starting at
            address tmp
    }

    // permutation P
    for(int i=0; i<32; i++)
        set_bit(out, i, get_bit(tmp, P[i] - 1));
        // out[i] = tmp[P[i] - 1]
}
```

2.5 Feistel Structure

As previously defined, we require 2 feistel networks, 1 for encryption and 1 for decryption. They are implemeted as follows:
Encrption:

```
/*
       |                                 |
       v                                 v
  +--------------------------------------------+
  |      L               |            R        |
  +--------------------------------------------+
       |                                 |
       |                 +---------------|---------- round key
       |                 |               |
       |                 v               |
       |           +---------+           |
       _           |         |           |
      |+|<---------|    F    |<----------+
       -           |         |           |
       |           +---------+           |
       \                                 /
        \_______________________________,
        /\
       |  \------------------------------+
       |                                 |
       |                                 |
       v                                 v
  +-------------------------------------------+
  |      L'                |            R'     |
  +-------------------------------------------+
       |                                 |
       v                                 v

1 round feistel structure (encryption) */
void feistel1(unsigned char* l, unsigned char* r, int round)
{
    // f_out
    unsigned char f_out[4];
    f(r, f_out, round);

    // save  R
    unsigned char r_back_up[4];
    for(int i=0; i<4; i++) r_back_up[i] = r[i];

    // R'
    for(int i=0; i<4; i++) r[i] = l[i] ^ f_out[i];

    // L'
    for(int i=0; i<4; i++) l[i] = r_back_up[i];
}
```

Decryption:

```
/*
       |                                 |
       v                                 v
  +---------------------------------------------+
  |      L                  |          R         |
  +---------------------------------------------+
       |                                 |
       |                  +--------------|---------- round key
       |                  |              |
       |                  v              |
       |            +---------+          |
       |            |         |          -
       |------------>|    F    |---------->|+|
       |            |         |          -
       |            +---------+          |
       \                                 /
        \_______________________________,
        /\
       |  \------------------------------+
       |                                 |
       |                                 |
       v                                 v
  +-------------------------------------------+
  |      L'                |            R'     |
  +-------------------------------------------+
       |                                 |
       v                                 v

       1 round feistel structure (decryption) */
void feistel2(unsigned char* l, unsigned char* r, int round)
{
    // f_out
    unsigned char f_out[4];
    f(l, f_out, round);

    // save  L
    unsigned char l_back_up[4];
    for(int i=0; i<4; i++) l_back_up[i] = l[i];

    // L'
    for(int i=0; i<4; i++) l[i] = r[i] ^ f_out[i];

    // R'
    for(int i=0; i<4; i++) r[i] = l_back_up[i];
}
```

2.6 Key Scheduling

The key scheduling procedure takes an extra flag “reversed” which indicates whether to store the round keys in the revese order (required for decryption). The algorithm can be implemented using the bit-access functions defined earlier and hard-coded tables for PC-1 and PC-2:

```
void key_schedule(unsigned char* key, int reversed)
{
    unsigned char curr_key[56];
    unsigned char tmp_key[56];
    zero(curr_key, 7);

    // PC-1
    for(int i=0; i<56; i++)
        set_bit(curr_key, i, get_bit(key, PC_1[i] - 1));

    printf("after PC-1:\n");
    for(int i=0; i<7; i++)
        printf("%02x ", curr_key[i]);
    printf("\n\n");

    for(int i=0; i<16; i++)
    {
        // shift = 1 or 2
        int shift = get_shift(i);
        zero(tmp_key, 7);

        // rotate C
        for(int j=0; j<28; j++)
        {
            int from = j + shift; from = from % 28;
            set_bit(tmp_key, j, get_bit(curr_key, from));
        }

        // rotate D
        for(int j=0; j<28; j++)
        {
            int to = j + 28;
            int from = j + shift; from = from % 28 + 28;
            set_bit(tmp_key, to, get_bit(curr_key, from));
        }

        // copy to curr_key
        for(int j=0; j<7; j++) curr_key[j] = tmp_key[j];

        printf("CD: %d\n", i);
        for(int j=0; j<7; j++)
            printf("%02x ", curr_key[j]);
        printf("\n\n");

        // PC-2
        int idx = i;
        if(reversed) idx = 15 - idx;
        zero(round_keys[idx], 6);
        for(int j=0; j<48; j++)
            set_bit(round_keys[idx], j, get_bit(curr_key, PC_2[j]));
    }

}
```

2.7 Combined

```
// encrypt 64 bits starting at address "block"
void des_enc(unsigned char* block, unsigned char* key, unsigned char* out)
{
    key_schedule(key, 0);

    // out is initialized to the input block and changed in place
    for(int i=0; i<8; i++) out[i] = block[i];

    initial_permutation(out);

    for(int i=0; i<16; i++)
    {
        feistel1(out, &out[4], i);
    }

    final_permutation(out);
}

// decrypt 64 bits starting at address "block"
void des_dec(unsigned char* block, unsigned char* key, unsigned char* out)
{
    key_schedule(key, 1);

    // out is initialized to the input block and changed in place
    for(int i=0; i<8; i++) out[i] = block[i];

    initial_permutation(out);

    for(int i=0; i<16; i++)
    {
        feistel2(out, &out[4], i);
    }

    final_permutation(out);
}
```

2.8 OFB

The OFB circuit for decryption and encryption are similar.
Encryption:

```
/*
  +-------------+
  |      IV     |------------.       .--------------.       .--------------.       .-    ...
  +-------------+           |        |             |        |             |        |
        |                   v        |             v        |             v        |
        |            +-------------+ |      +-------------+ |      +-------------+ |
        |            |             | |      |             | |      |             | |
        |            |     DES     | |      |     DES     | |      |     DES     | |
        |            |             | |      |             | |      |             | |
        |            +-------------+ |      +-------------+ |      +-------------+ |
        |                   |        |             |        |             |        |
        |                   |        |             |        |             |        |
        |                   |________'             |________'             |________'
        |                   _                      _                      _
        |         .------->|+|           .------->|+|           .------->|+|
        |         |         -            |         -            |         -
        |          |          |            |          |            |          |
        v          m0         |            m1         |            m2         |
                              v                       v                       v
                             c0                      c1                      c2
 */
void ofb_enc(unsigned char* plaintext, int len, unsigned char* key, unsigned char* iv,
    unsigned char* ciphertext)
{
    unsigned char in[8];
    unsigned char out[8];
    for(int i=0; i<8; i++) in[i] = iv[i];

    int idx = 0;
    for(int i=0; i<len/8; i++)
    {
        des_enc(in, key, out);
        for(int j=0; j<8; j++)
        {
            ciphertext[idx] = out[j] ^ plaintext[idx];
            idx++;
            in[j] = out[j]; // next input
        }
    }
}
```

Decryption:

```
/*

  +------------+
  |     IV     |------------.        .-------------.        .-------------.        .-   ...
  +------------+            |        |             |        |             |        |
        |                   v        |             v        |             v        |
        |             +-----------+  |       +-----------+  |       +-----------+  |
        |             |           |  |       |           |  |       |           |  |
        |             |    DES    |  |       |    DES    |  |       |    DES    |  |
        |             |           |  |       |           |  |       |           |  |
        |             +-----------+  |       +-----------+  |       +-----------+  |
        |                   |        |             |        |             |        |
        |                   |        |             |        |             |        |
        |                   |--------'             |--------'             |--------'
        |                   _                      _                      _
        |          .------->|+|            .------>|+|            .------>|+|
        |          |        -              |       -              |       -
        |          |        |              |       |              |       |
        v          c0       |              c1      |              c2      |
                            v                      v                      v
                           m0                     m1                     m2
*/
void ofb_dec(unsigned char* ciphertext, int len, unsigned char* key, unsigned char* iv,
    unsigned char* decryptedtext)
{
    unsigned char in[8];
    unsigned char out[8];
    for(int i=0; i<8; i++) in[i] = iv[i];

    int idx = 0;
    for(int i=0; i<len/8; i++)
    {
        des_enc(in, key, out);
        for(int j=0; j<8; j++)
        {
            decryptedtext[idx] = out[j] ^ ciphertext[idx];
            idx++;
            in[j] = out[j]; // next input
        }
    }
}
```

2.9 Input/Output

The program takes input plaintext or ciphertext from files "plaintext.txt" and "ciphertext" respectively and key from "key". During encryption the plaintext is read, the IV is generated the passed to the OFB circuit. The generated IV is output first and then the ciphertext.
During decryption the IV must be read first (first 8 bytes) and then the rest of the ciphertext.