SCHOOL OF ELECTRICAL ENGINEERING, UNIVERSITY OF THE WITSWATERSRAND

ELEN3015 - Data and Information Management

Lecturer: Ling Cheng

Lab 2 Data Encryption Algorithm

20 April 2018

Tyson Cross 1239448

***Abstract*— This report describes the implementation of the DES (Data Encryption Standard) in MATLAB. A function to generate a 56-bit key, and a key-schedule of 16 48-bit sub-keys is detailed, along with an implementation of the 16 Fiestel cipher rounds of the permutation, substitution and subkey splitting and interleaving in the DEA. The successful encryption and decryption of a message is demonstrated, and the strengths and weaknesses of the specific implementation and of DES are general are discussed.**

INTRODUCTION

The DES is an symmetric-key, block cryptographic system designed by the United States' National Security Agency (NSA) for the secure and secret exchange of unclassified messages[1]. Adopted from IBM's *Lucifer* project, the final standard was first published in 1977[2], and has subsequently been replaced by a newer standard called AES. DES's key is processed to produce 16 subkeys, with each subkey being used once as input (along with the message being enciphered) into a Fiestel cipher. The interative block encryption system consists of a 16-round schedule of permutation and substitution operations, along with XOR operations, to swap and mix the subkeys and message around to produce the encrypted output. The general structure of DES is shown

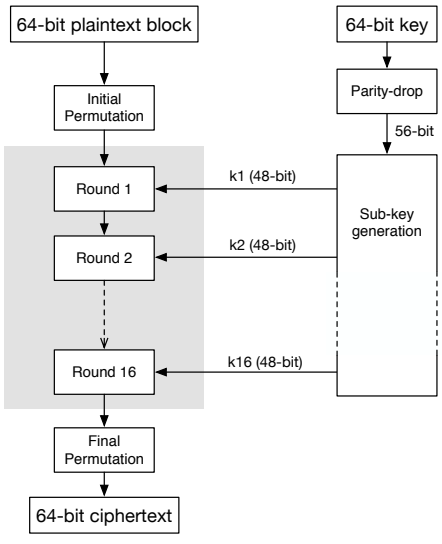

Fig. 1: Data Encryption System overview

in Figure 1. Several of the substitution operations are non-linear. The algorithm is reversible, so the output ciphertext can be fed back into the algorithm with the reverse subkey schedule, to reproduce the original plain text message. The DES is now considered insecure, and within computational feasibility to break using differential analysis and brute force attacks. The use of triple DES is still considered a reasonably secure method[1], as the increased keyspace in TDES makes the system more resistant to BFA, as will be discussed in detail later in the report.

1 DES KEY SCHEDULE

The DES specifies a 64-bit key, of which only 56-bits are used to generate the subkeys. These 56-bits are chosen, and permuted, according to a parity-drop lookup table, shown in Table I, which is read as a simple map for each entry (in the original 64-bit key) starting at the top left and read along to the right row-wise. The discarded bits are usually used as a parity check pre-encryption. In the implementation presented,

TABLE I: Key parity-drop table

57	49	41	33	25	17	09	01
58	50	42	34	26	18	10	02
59	51	43	35	27	19	11	03
60	52	44	36	63	55	47	39
31	23	15	07	62	54	46	38
30	22	14	06	61	53	45	37
29	21	13	05	28	20	12	04

this formal stage of the key generation is not used, and an input 56-bit key input is directly processed to create subkeys without the parity/permutation stage. The 56-bit key is then split into two halves of 28-bits. These halves are split using the function `splitter` shown in Code Listing 6 in Appendix I). Each half is then bit-shifted to the left, a number of places depending on the round key schedule, shown in Table II. The function to accomplish this shift is shown in Code Listing 7 in Appendix I. The two halves are then fed into a permutation function, acting as a compression D-box, which produces a 48-bits output subkey. The compression table used to permute and discard the extra bits is shown in Table III. The permutation function

TABLE II: Subkey shifting schedule

Rounds	Shift
1, 2, 9, 16	1 bit
Others	2 bits

is implemented as general function with a switch statement to swap between hard-coded permutation tables as shown in Code Listing 8 in Appendix I. The function argument to specify key compression permutation is `'compression'`. The resulting 16

TABLE III: Key compression table

14	17	11	24	01	05	03	28
15	06	21	10	23	19	12	04
26	08	16	07	27	20	13	02
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

subkeys act as an additional inputs which are are combined cryptographically with the message during the encipherment for each individual round.

2 Initial and final permutations

Before the 16 feistel cipher rounds, there is an initial permutation, which does not require a key, and has no cryptographic purpose, apart from diffusing the plaintext. After the encryption rounds are complete, there is also a another fixed keyless permutation. It has been speculated that the purpose of these two permutations are to restrict an historic software implementation of the algorithm, as it was initially designed for hardware implementation. These permutations are implemented as hardcoded blocks for the general purpose `permuter` function, shown in Code Listing 8 in Appendix I.

3 Feistel cipher round

A diagram of a single DES round is shown in Figure 2. A single round of encryption proceeds as follows: first the input block is split into two 32-bit halves. The left half, L is expanded with an expansion permutation D-box to be 48-bits. This 48-bit block is then XORed with the input 48-bit subkey for the round. This register is then fed into a non-linear substitution box, which both substitutes values and reduces the block back to 32-bits. A straight permutation then occurs, and the processed L block is recombined with the original R half, but swapped. Each round is called with the `DES` function shown in Code Listing 10 in Appendix I.

3.a Expansion

The expansion permutation expands every 4 bits (a nibble) into 6 bits. The first bit of the new 6-bits comes

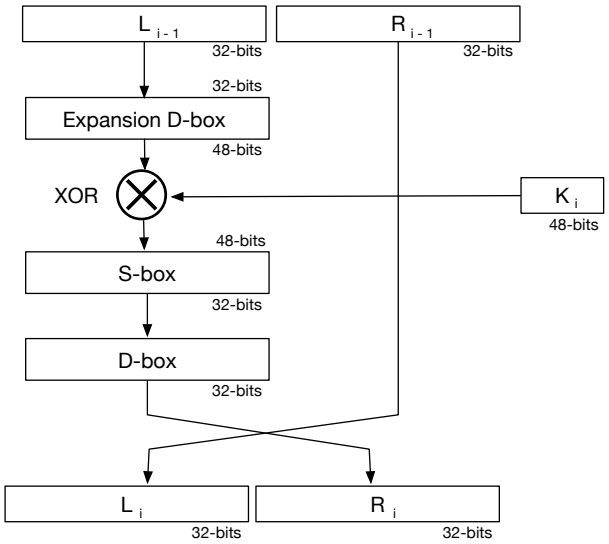

Fig. 2: A single DES Fiestel round

from the last bit of the preceding section (wrapping around the entire 32-bit sequence). So the first bit in the expanded sequence is the 32nd bit from the original sequence. The 6th bit in each new subsection is the first bit in the following sequence. This process results in a 48-bit sequence. The implementation uses the general `permuter` function, shown in Code Listing 8 in Appendix I.

3.b XORing with the round subkey

The resulting 48-bit sequence is then XORed with the subkey for the block. This value is then fed into a substitution box.

3.c S-box

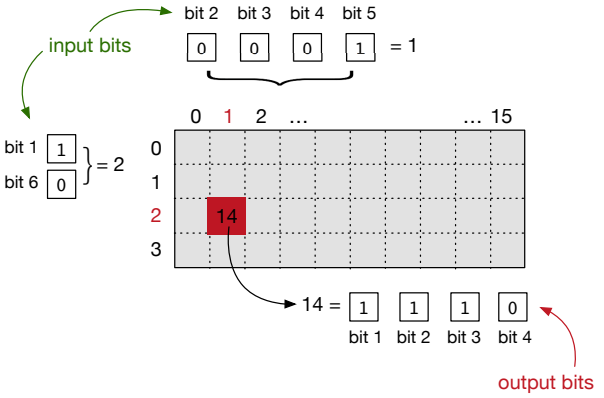

Fig. 3: S-box indexing

The 48-bit block resulting from the XOR operation with the subkey is now disguised and compressed, through the use of a non-linear substitution box. The input sequence takes every 6-bits, and recombines them into two coordinate values which are used to read a resulting value in the s-box. The 4-bit binary value of the entry at the coordinate index is then

used as the bit value for that section. This process is demonstrated in Figure 3, and implemented in the function `substitution`, shown in Code Listing 9 in Appendix I. The output is a 32-bit block.

3.d D-box

These 32 bits then have a straight permutation, from the `permuter` function, using the `pbox` switch case, which further mixes about the bit values.

3.e Swapping

Finally the output half is recombined with the original *R* half, swapped so the output processed block becomes the new right half. In the final and 16th round, this processed is either skipped, or repeated one more time. This operation ensures that the algorithm is reversible (so that the output ciphertext can be fed back into the algorithm with a reverse key-schedule to produce the original plaintext.)

4 DES ANALYSIS

After 16 rounds of encipherment, the extra swapping of the two halves of the message, and the final permutation, the encryption is complete. with a 64-bit output block of ciphertext from the original 64-bit input plaintext. DES results in good diffusion and confusion, with a strong avalanche effect, meaning that minor changes in plaintext lead to large changes in the ciphertext. In addition, there is substantial dependence for each individual bit in the ciphertext on multiple bits in the plaintext. The non-linearity of the s-box design means that DES is resistant to simplistic cryptanalysis[3], but as [4] have shown, DES is vulnerable to differential cryptanalysis.

4.a Key strength

TABLE IV: Weak DES Keys

0101 0101 0101 0101
FEFE FEFE FEFE FEFE
E0E0 E0E0 F1F1 F1F1
1E1E 1E1E 0F0F 0F0F

Out of the possible 2^{56} keys, there are four *weak* keys for DES, which result in very poor or no encryption. The keys shown in Table IV list the four *weak* keys which will only result in a single unique subkey (i.e all the subkeys will be identical). Using any of these keys would result in the plaintext being decrypted back from the ciphertext every two applications of the Feistel cipher. There are also six *semi-weak* keys, which only have 2 unique subkeys, and several *possibly-weak* keys which only result in 4 unique subkeys. These keys are examined and analysed using the code for Question 2, shown in Code Listing 1 in Appendix I.

4.b Breaking DES

DES has been shown to be insecure from several demonstrated attacks. With a dependant key schedule and a chosen plaintext, the full 16-round DES has been broken in 2^{47} attempts. With a known plaintext and completely independent subkeys, DES can be broken within 2^{61} attempts. DES can be broken using the birthday paradox attack to make changes from both ends (both the plaintext and the ciphertext), which involves 2^{56} attempts. In 1999, DES was broken in less than 24 hours by a large cluster of distrusted personal computers through Distributed.Net. "Double-DES", in which ciphertext is encrypted twice, with two seperate keys, can still be broken within 2^{57} attempts, a very marginal an insufficient improvement. A more secure way to use DES today is what it termed Triple DES, due to the use of three separate keys, and three full 16-rounds of encryption. The resulting schedule looks like :

$$\text{ciphertext} = \mathrm{E}_{K3}\big(\mathrm{D}_{K2}\big(\mathrm{E}_{K1}\big(\text{plaintext}\big)\big)\big)$$

This can be considered an encryption scheme with a 168-bit key, but due to the continuing vulnerability of a meet-in-the middle attack, only requires $2^{2x56} = 2^{112}$ attempts to break. This is a large improvement on normal DES, but considered only marginally secure and still vulnerable to breaking with modern consumer-level computing hardware.

5 IMPLEMENTATION ANALYSIS

The implementation presented is programmed with functional programming, and is only appropriate in an educational context for demonstration and analysis of the DES for historic purposes and greater understanding of the Fiestel function's workings. A test message was encrypted using the implementation of DES, shown in Question 4, Code Listing 3 in Appendix I, which was successfully decrypted by putting the cipher text back into the 16-rounds, and reversing the keyschedule.

CONCLUSION

DES was implemented in Mathwork's MATLAB using functional programming. The DEA was explained with reference to the implementation. The weakness of specific input keys was discussed, and a demonstration message was encrypted and decrypted with example code.

REFERENCES

[1] W. Stallings, *Cryptography and network security: principles and practice*. Pearson Education India, 2003.

[2] FIPS, "Data encryption standard (des)," *FIPS PUB*, pp. 46–3, 1999.

[3] N. Nedjah and L. d. M. Mourelle, "Designing substitution boxes for secure ciphers," *International Journal of Innovative Computing and Applications*, vol. 1, no. 1, pp. 86–91, 2007.

[4] E. Biham and A. Shamir, *Differential cryptanalysis of the data encryption standard*. Springer Science & Business Media, 2012.

QUESTION 2

Code 1: question2.m

```
% ELEN3015 Lab 2, Question 2

clc
clear all

% keys to analyse
key1 = '1F1F1F1F0E0E0E0E';
key2 = '1FFE1FFE0EFE0EFE';
key3 = '1FFEFE1F0EFEFE0E';

% convert to binary row vectors
key_bin1 = hex2binary(key1);
key_bin2 = hex2binary(key2);
key_bin3 = hex2binary(key3);

% initial permutation
key_string1 = permuter(key_bin1, 'parity');
key_string2 = permuter(key_bin2, 'parity');
key_string3 = permuter(key_bin3, 'parity');

% DEA rounds to generate all subkeys
for round_no=1:16
    key_schedule1(round_no,:) = generateSubKey(key_string1,round_no);
    key_schedule2(round_no,:) = generateSubKey(key_string2,round_no);
    key_schedule3(round_no,:) = generateSubKey(key_string3,round_no);
end

% count and classify the subkeys
[count1, classification1] = analyseSubKeys(key_schedule1);
[count2, classification2] =  analyseSubKeys(key_schedule2);
[count3, classification3] =  analyseSubKeys(key_schedule3);

% output
disp(['Key ', key1, ' has ', num2str(count1), ' unique subkey(s). It is a ', classification1, ' key'])
disp(['Key ', key2, ' has ', num2str(count2), ' unique subkey(s). It is a ', classification2, ' key'])
disp(['Key ', key3, ' has ', num2str(count3), ' unique subkey(s). It is a ', classification3, ' key'])
```

When question2.m is run in the workspace, the following output is displayed in the command window:

```
  Key 1F1F1F1F0E0E0E0E has 1 unique subkey(s). It is a weak key
Key 1FFE1FFE0EFE0EFE has 2 unique subkey(s). It is a semi weak key
Key 1FFEFE1F0EFEFE0E has 4 unique subkey(s). It is a possibly weak key
```

QUESTION 3

Code 2: question3.m

```
% ELEN3015 Lab 2, Question 3

clc
clear all

% inputs
plaintext_str = '0000000100100011010001010110011110001001101010111100110111101111';
key_str       = '0001001100110100010101110111100110011011101111001101111111110001';
plaintext = plaintext_str - '0';
key_64 = key_str - '0';
key_56 = permuter(key_64, 'parity');                    % discard parity bits and permute
block = permuter(plaintext,'initial');                  % initial permutation

round_no = 1;
subkey = generateSubKey(key_56,round_no);

% perform a DEA round of encryption
[ L_block, R_block ] = DES(block, round_no, subkey);

% output
disp(['Input 64-bit message: ', binary2hex(plaintext),]);
disp(['Input 64-bit key:     ', binary2hex(key_64)])
disp(['Permuted 56-bit key:    ', binary2hex(key_56)])
disp(['48-bit round (', num2str(round_no),') subkey:  ', binary2hex(subkey)])
disp(['Permuted block: ', num2string(block)])
disp(['Left: ', num2string(L_block), '   Right: ', num2string(R_block)])
```

When question3.m is run in the workspace, the following output is displayed in the command window:

```
Input 64-bit message: 0123456789ABCDEF
Input 64-bit key:     133457799BBCDFF1
Permuted 56-bit key:    F0CCAAF556678F
48-bit round (1) subkey:  1B02EFFC7072
Permuted block: 1100110000000000110011001111111111110000101010101111000010101010
Left: 11110000101010101111000010101010   Right: 11101111010010100110010101000100
```

QUESTION 4

Code 3: question4.m

```
% ELEN3015 Lab 2, Question 4

clc
clear all

% inputs
plaintext_str = '0123456789ABCDEF';                         % test block
plaintext = hex2binary(plaintext_str);                      % convert to binary row vector
key_str = '0001001100110100010101110111100110011011101111001101111111110001';
key_64 = key_str - '0';                                     % convert to binary row vector
key_56 = permuter(key_64, 'parity');                        % discard parity bits and permute

%% encryption
block = permuter(plaintext,'initial');                      % initial permutation
for round_no = 1:16                                         % 16 rounds of DES
    subkey = generateSubKey(key_56,round_no);               % subkey generation for each round
    [ L, R ] = DES(block, round_no, subkey);                % DEA (encryption)
    block = [L R];                                          % rejoin the L and R halves
end
cipherblock = permuter(block, 'final');                     % final permutation


%% decryption
out_block = permuter(cipherblock,'initial');                % initial permutation
for round_no=1:16                                           % 16 rounds of DES
    subkey = generateSubKey(key_56,17-round_no);            % subkeys generated in reverse order
    [L, R] = DES(out_block, round_no, subkey);              % DEA (decrpytion)
    out_block = [L R];                                      % rejoin block halves
end
decrypted = permuter(out_block, 'final');                   % final permutation


% output and check
decrypted_str = binary2hex(decrypted);
cipher_str = binary2hex(cipherblock);
disp(['Input 64-bit key:     ', binary2hex(key_64)])
disp(['Encryted ciphertext:  ', cipher_str])
disp(['Original input text:  ', plaintext_str]);
disp(['Decrypted plaintext:  ', decrypted_str])
disp(' ')

% check
if isequal(decrypted_str,plaintext_str)
    disp('The decrypted block matches the plaintext block');
else
    warning('The decrypted block does NOT match the plaintext block');
end
```

When question4.m is run in the workspace, the following output is displayed in the command window:

```
Input 64-bit key:     133457799BBCDFF1
Encryted ciphertext:  85E813540F0AB405
Original input text:  0123456789ABCDEF
Decrypted plaintext:  0123456789ABCDEF

The decrypted block matches the plaintext block
```

Code 4: generateSubKey.m

```
function [ outKey ] = generateSubKey( inKey, i )
% generateSubKey() produces a DES-compliant 48-bit key from an input 56-bit key
% for a specific round number

if i>16 || i<1
    error('Round number not in valid range [1-16]')
end

if length(inKey)≠56
    error('Please provide a 56-bit key')
end

% Take the 56-bit key, split into 2 subkeys
% Each subkey half undergoes 16 rounds of subkey generation:
%    a) bitshift each half by (1 or 2) bits, on a fixed schedule
%    b) combine halves, compress down to a 48-bit key which is stored.

[keyL, keyR] =  splitter(inKey);
for round_no=1:i
keyL = shiftKey(keyL,round_no);
keyR = shiftKey(keyR,round_no);
end

outKey = permuter([keyL keyR],'compression');

end
```

Code 5: analyseSubKeys.m

```
function [count, classification] = analyseSubKeys( key_schedule )
% analyseSubKeys counts and classifies the strength of a subkey schedule

% determine number of unique keys
[n, m] = size(unique(key_schedule,'rows'));
count = n;

% classify the input keys
switch n
    case 1
        classification = 'weak';
    case 2
        classification = 'semi weak';
    case 4
        classification = 'possibly weak';
    otherwise
        classification = 'flat/linear';
end
end
```

Code 6: splitter.m

```
function [ outKey_l, outKey_r ] = splitter(inKey)
% splitter() splits an input row vector in half, into two seperate row vectors of equal length

if mod(length(inKey),2)
    error('Block length must be even')
else len = length(inKey)/2;
end

outKey_l = inKey(1:len);
outKey_r = inKey(len+1:end);
end
```

Code 7: shiftKey.m

```
function [ outKey ] = shiftKey( inKey, round_no)
% Left-shifts (circularly) key entries, 1 or 2 places, depending on the round

if (round_no==1 || round_no==2 || round_no==9 || round_no==16)
    shift = -1;
else shift = -2;
end

outKey = circshift(inKey',[shift,0])';
end
```

Code 8: permuter.m

```
function [ outBlock ] = permuter(inBlock, mode)
% Performs fixed permutation upon input block (64-/56-/48-bit)

% The permutation uses fixed D-tables and size of i/o block,
% determined by the mode argument:
% For blocks: 'initial'(64-bit), 'final'(64-bit), 'expansion' (32->48-bit) or 'pbox'(32-bit)
% For keys: 'parity' (64->56-bit) or 'compression' (56->48-bit)

switch mode
    case 'initial'
        len = 64;
        permutationTable = [58, 50, 42, 34, 26, 18, 10, 02, 60, 52, 44, 36, 28, 20, 12, 04,...
                            62, 54, 46, 38, 30, 22, 14, 06, 64, 56, 48, 40, 32, 24, 16, 08,...
                            57, 49, 41, 33, 25, 17, 09, 01, 59, 51, 43, 35, 27, 19, 11, 03,...
                            61, 53, 45, 37, 29, 21, 13, 05, 63, 55, 47, 39, 31, 23, 15, 07];
    case 'final'
        len = 64;
        permutationTable = [40, 08, 48, 16, 56, 24, 64, 32, 39, 07, 47, 15, 55, 23, 63, 31,...
                            38, 06, 46, 14, 54, 22, 62, 30, 37, 05, 45, 13, 53, 21, 61, 29,...
                            36, 04, 44, 12, 52, 20, 60, 28, 35, 03, 43, 11, 51, 19, 59, 27,...
                            34, 02, 42, 10, 50, 18, 58, 26, 33, 01, 41, 09, 49, 17, 57, 25];

    case 'parity'
        len = 56;
        permutationTable = [57, 49, 41, 33, 25, 17, 09, 01, 58, 50, 42, 34, 26, 18,...
                            10, 02, 59, 51, 43, 35, 27, 19, 11, 03, 60, 52, 44, 36,...
                            63, 55, 47, 39, 31, 23, 15, 07, 62, 54, 46, 38, 30, 22,...
                            14, 06, 61, 53, 45, 37, 29, 21, 13, 05, 28, 20, 12, 04];
    case 'compression'
        len = 48;
        permutationTable = [14, 17, 11, 24, 01, 05, 03, 28, 15, 06, 21, 10,...
                            23, 19, 12, 04, 26, 08, 16, 07, 27, 20, 13, 02,...
                            41, 52, 31, 37, 47, 55, 30, 40, 51, 45, 33, 48,...
                            44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29, 32];
     case 'expansion'
         len = 48;
        permutationTable = [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, 30, 31, 32, 01];
    case 'pbox'
        len = 32;
        permutationTable = [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];
    otherwise
        error('Specify permutation mode: ''initial'', ''final'', ''parity'', ''compression'', ...
            ''expansion'', or ''pbox'''')
end

outBlock = zeros(1,len);

for iter=1:len
    outBlock(iter) = inBlock(permutationTable(iter));
end

end
```

Code 9: substitution.m

```
function [ outBlock ] = substitution( inBlock )
% Performs non-linear S-Box substitution on inBlock

substitutionTable =...
    [14, 04, 13, 01, 02, 15, 11, 08, 03, 10, 06, 12, 05, 09, 00, 07;...
     00, 15, 07, 04, 14, 02, 13, 01, 10, 06, 12, 11, 09, 05, 03, 08;...
     04, 01, 14, 08, 13, 06, 02, 11, 15, 12, 09, 07, 03, 10, 05, 00;...
     15, 12, 08, 02, 04, 09, 01, 07, 05, 11, 03, 14, 10, 00, 06, 13];

substitutionTable(:,:,2) =...
    [15, 01, 08, 14, 06, 11, 03, 04, 09, 07, 02, 13, 12, 00, 05, 10;...
     03, 13, 04, 07, 15, 02, 08, 14, 12, 00, 01, 10, 06, 09, 11, 05;...
     00, 14, 07, 11, 10, 04, 13, 01, 05, 08, 12, 06, 09, 03, 02, 15;...
     13, 08, 10, 01, 03, 15, 04, 02, 11, 06, 07, 12, 00, 05, 14, 09];

substitutionTable(:,:,3) =...
    [10, 00, 09, 14, 06, 03, 15, 05, 01, 13, 12, 07, 11, 04, 02, 08;...
     13, 07, 00, 09, 03, 04, 06, 10, 02, 08, 05, 14, 12, 11, 15, 01;...
     13, 06, 04, 09, 08, 15, 03, 00, 11, 01, 02, 12, 05, 10, 14, 07;...
     01, 10, 13, 00, 06, 09, 08, 07, 04, 15, 14, 03, 11, 05, 02, 12];

substitutionTable(:,:,4) =...
    [07, 13, 14, 03, 00, 06, 09, 10, 01, 02, 08, 05, 11, 12, 04, 15;...
     13, 08, 11, 05, 06, 15, 00, 03, 04, 07, 02, 12, 01, 10, 14, 09;...
     10, 06, 09, 00, 12, 11, 07, 13, 15, 01, 03, 14, 05, 02, 08, 04;...
     03, 15, 00, 06, 10, 01, 13, 08, 09, 04, 05, 11, 12, 07, 02, 14];

substitutionTable(:,:,5) =...
    [02, 12, 04, 01, 07, 10, 11, 06, 08, 05, 03, 15, 13, 00, 14, 09;...
     14, 11, 02, 12, 04, 07, 13, 01, 05, 00, 15, 10, 03, 09, 08, 06;...
     04, 02, 01, 11, 10, 13, 07, 08, 15, 09, 12, 05, 06, 03, 00, 14;...
     11, 08, 12, 07, 01, 14, 02, 13, 06, 15, 00, 09, 10, 04, 05, 03];

substitutionTable(:,:,6) =...
    [12, 01, 10, 15, 09, 02, 06, 08, 00, 13, 03, 04, 14, 07, 05, 11;...
     10, 15, 04, 02, 07, 12, 09, 05, 06, 01, 13, 14, 00, 11, 03, 08;...
     09, 14, 15, 05, 02, 08, 12, 03, 07, 00, 04, 10, 01, 13, 11, 06;...
     04, 03, 02, 12, 09, 05, 15, 10, 11, 14, 01, 07, 06, 00, 08, 13];

substitutionTable(:,:,7) =...
    [04, 11, 02, 14, 15, 00, 08, 13, 03, 12, 09, 07, 05, 10, 06, 01;...
     13, 00, 11, 07, 04, 09, 01, 10, 14, 03, 05, 12, 02, 15, 08, 06;...
     01, 04, 11, 13, 12, 03, 07, 14, 10, 15, 06, 08, 00, 05, 09, 02;...
     06, 11, 13, 08, 01, 04, 10, 07, 09, 05, 00, 15, 14, 02, 03, 12];

substitutionTable(:,:,8) =...
    [13, 02, 08, 04, 06, 15, 11, 01, 10, 09, 03, 14, 05, 00, 12, 07;...
     01, 15, 13, 08, 10, 03, 07, 04, 12, 05, 06, 11, 00, 14, 09, 02;...
     07, 11, 04, 01, 09, 12, 14, 02, 00, 06, 10, 13, 15, 03, 05, 08;...
     02, 01, 14, 07, 04, 10, 08, 13, 15, 12, 09, 00, 03, 05, 06, 11];

outBlock = [];

for i=1:8
    row_index_1 = num2string(inBlock(i*6-5));               % first bit of the 6-bit block
    row_index_2 = num2string(inBlock(i*6));                 % last bit of the 6-bit block
    row_index = strcat(row_index_1,row_index_2);            % convert the bits into an index
    row = bin2dec(row_index) + 1;                           % matlab indexing starts at 1, not 0
    col = bin2dec(dec2bin(inBlock(i*6-4:i*6-1))') + 1;      % middle four bits as index
    value = dec2bin(substitutionTable(row,col,i),4)-'0';    % apply the sunstitution table
    outBlock = [outBlock value];                            % append the bit-block
end
```

Code 10: DES.m

```
function [ L, R ] = DES( inBlock, round_no, subKey)
% Performs encryption on an input 64-bit block, returns two encrypted 32-bit blocks

% if length(inBlock) < 64
%     inBlock = padarray(inBlock',64-length(inBlock),'post')';
% end

% Round
[L, R] = splitter(inBlock);                     % evenly split the block into two 32-bit halves

temp = R;                                       % store the L half value
R = permuter(R,'expansion');                    % perform the expansion permutation
R = xor(R, subKey);                             % XOR with the round subkey
R = substitution(R);                            % perform non-linear s-box substitution
R = permuter(R,'pbox');                         % perform the p-box permutation
R = xor(R,L);                                   % XOR with the original L half
L = temp;                                       % assign L to the output from the Feistel cipher
if round_no == 16                               % swap the L and R halves on the final round
    [R,L] = deal(L,R);
end

%% output
% disp(['Round ', num2str(round_no), ':'])
% disp(['L: ', num2string(L), '   R: ',num2string(R)])
end
```

Code 11: addParityBits.m

```
function [ output ] = addParityBits( bin_array )
% addParityBits counts and adds parity bits to a 56-bit key, producing a 64-bit key

if length(bin_array)≠56
    error('Please provide a 56-bit key')
end

bin_matrix = reshape(bin_array,[7,8])';
bin_matrix = [bin_matrix zeros(8,1)];

for i=1:8
    sum=0;
    for j=1:7
        sum = sum + bin_matrix(i,j);
    end
    bin_matrix(i,8)= 1-mod(sum, 2);
end

output = reshape(bin_matrix,1,64);

end
```