Data Encryption Standard

TianxiaoHu 14300240007

November 2, 2016

1 Experiment Background

The Data Encryption Standard is a symmetric-key algorithm for the encryption of electronic data. Although now considered insecure, it was highly influential in the advancement of modern cryptography. In recent years, the cipher has been superseded by the Advanced Encryption Standard (AES). Furthermore, DES has been withdrawn as a standard by the National Institute of Standards and Technology (formerly the National Bureau of Standards). The following report will describe the encryption steps in details, including each step's output. The date of today '20161102' is taken as an input example and the key is '0123456789abcdef'.

2 Experiment Enviornment

Processor: Intel(R) Core(TM) i5-4590 CPU@3.30GHz 3.30GHz
RAM: 8.00GB
System: Windows 10
Python interpreter: Anaconda 2.7.12

3 Simulation of DES in Details

3.1 Encryption

3.1.1 Get Keys

Load packaged needed.

```
In [1]: import binascii
        import numpy as np
```

Format the output so that each row has 20 bits.

```
In [2]: def pretty_print(array):
            for i in range(len(array)):
                if i % 10 == 0 and i != 0:
                    print '  ',
                if i % 30 == 0 and i != 0:
                    print
                print array[i],
```

Input keys string: 0123456789abcdef.

```
In [3]: keyhexstr = '0123456789abcdef'
```

Convert hexadecimal string into binary char array for convenience.

```
In [4]: def keyhexstr2keybinarr(keyhexstr):
            keybinstr = '0'*(4*len(keyhexstr)-len(bin(int(keyhexstr, 16)))+2)\
                        + bin(int(keyhexstr, 16)).replace('0b', '')
            keybinarr = []
            for i in range(len(keybinstr)):
                keybinarr.append(int(keybinstr[i]))
            return np.array(keybinarr)

        keybinarr = keyhexstr2keybinarr(keyhexstr)
```

Print he binary array.

```
In [5]: pretty_print(keybinarr)
```

```
0 0 0 0 0 0 0 1 0 0    1 0 0 0 1 1 0 1 0 0    0 1 0 1 0 1 1 0 0 1
1 1 1 0 0 0 1 0 0 1    1 0 1 0 1 0 1 1 1 1    0 0 1 1 0 1 1 1 1 0
1 1 1 1
```

Define the table which split and hash the key to A and B part (28×2).

```
In [6]: key_divide_table = [[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]]
        key_divide_table = np.array(key_divide_table) - 1
```

```
In [7]: key_A = map(lambda x: keybinarr[x], key_divide_table[0])
        key_B = map(lambda x: keybinarr[x], key_divide_table[1])
```

```
In [8]: pretty_print(key_A)
        print '\n---'
        pretty_print(key_B)
```

```
1 1 1 1 0 0 0 0 1 1    0 0 1 1 0 0 1 0 1 0    1 0 1 0 0 0 0 0
---
1 0 1 0 1 0 1 0 1 1    0 0 1 1 0 0 1 1 1 1    0 0 0 0 0 0 0 0
```

Concatenate two parts of key.

```
In [9]: key_unhashed = np.concatenate((key_A, key_B))
```

Reduce the 56-length key to 48-length.

```
In [10]: key_hash_table = [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]
         key_hash_table = np.array(key_hash_table) - 1

In [11]: key = map(lambda x: key_unhashed[x], key_hash_table)
```

Check the length.

```
In [12]: len(key)

Out[12]: 48

In [13]: pretty_print(key)
```

```
1 1 0 0 1 0 1 0 0 0    1 1 1 1 0 1 0 0 0 0    0 0 1 1 1 0 1 1 1 0
0 0 0 1 1 1 0 0 0 0    0 0 1 1 0 0 1 0
```

Define how many bits the key need to rotate left in the loops.

```
In [14]: key_loop_table = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]
```

Get 16 keys and save.

```
In [15]: def rotate_left(arr, n):
             return np.concatenate((arr[n:], arr[:n]))

In [16]: keys = []
         for i in range(16):
             global key_A, key_B
             key_A = rotate_left(key_A, key_loop_table[i])
             key_B = rotate_left(key_B, key_loop_table[i])
             key_unhashed = np.concatenate((key_A, key_B))
             key = map(lambda x: key_unhashed[x], key_hash_table)
             keys.append(key)
         keys = np.array(keys)
```

Keys are sorted in array `keys`.

```
In [17]: keys.shape

Out[17]: (16L, 48L)
```

3.1.2 IP Function

Input the plaintext.

```
In [18]: plaintext = '20161102'
```

For extension and convenience of decryption, use ASCII to encode the number.

```
In [19]: def asc2binarr(ascstr):
             binstr = bin(int(binascii.hexlify(ascstr), 16)).replace('0b', '')
             binstr = '0' * (8 - len(binstr) % 8) + binstr
             return np.array(map(lambda x: int(x), binstr))

         textbinarr = asc2binarr(plaintext)
```

Define the table of IP function.

```
In [20]: IP_table = [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]
         IP_table = np.array(IP_table) - 1

In [21]: pretty_print(textbinarr)
```

```
0 0 1 1 0 0 1 0 0 0    1 1 0 0 0 0 0 0 1 1    0 0 0 1 0 0 1 1 0 1
1 0 0 0 1 1 0 0 0 1    0 0 1 1 0 0 0 1 0 0    1 1 0 0 0 0 0 0 1 1
0 0 1 0
```

IP function.

```
In [22]: textbinarr_IP = np.array(map(lambda x: textbinarr[x], IP_table))

In [23]: pretty_print(textbinarr_IP)
```

```
0 0 0 0 0 0 0 0 1 1    1 1 1 1 1 1 0 0 0 0    1 0 0 0 0 0 1 1 0 1
0 0 0 0 0 0 0 0 0 0    1 1 1 1 1 1 1 1 0 0    0 0 0 0 0 0 1 0 0 0
1 0 0 1
```

3.1.3 The Feistel (F) function

Define the table for extending.

```
In [24]: text_extend_table = [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]
         text_extend_table = np.array(text_extend_table) - 1
```

Split the text into two parts.

```
In [25]: text_A = textbinarr_IP[: 32]
         text_B = textbinarr_IP[32: ]
```

Extend the right part.

```
In [26]: text_B_extend = np.array(map(lambda x: text_B[x], text_extend_table))

In [27]: pretty_print(text_B_extend)
```

```
1 0 0 0 0 0 0 0 0 0    0 1 0 1 1 1 1 1 1 1    1 1 1 0 1 0 0 0 0 0
0 0 0 0 0 1 0 1 0 0    0 1 0 1 0 0 1 0
```

XOR with key.

```
In [28]: text_B_xor = text_B_extend ^ keys[0]

In [29]: pretty_print(text_B_xor)
```

```
1 0 0 0 1 0 1 1 0 0    0 1 0 1 0 1 1 0 0 1    1 0 0 1 0 0 0 1 1 0
1 1 0 1 0 1 1 1 0 1    1 1 1 1 0 1 1 1
```

Shorten to 32 bits.

```
In [30]: text_B_xor = text_B_xor.reshape(8, 6)
```

Divide the text into 8×6 small pieces.

```
In [31]: text_B_xor

Out[31]: array([[1, 0, 0, 0, 1, 0],
                [1, 1, 0, 0, 0, 1],
                [0, 1, 0, 1, 1, 0],
                [0, 1, 1, 0, 0, 1],
                [0, 0, 0, 1, 1, 0],
                [1, 1, 0, 1, 0, 1],
                [1, 1, 0, 1, 1, 1],
                [1, 1, 0, 1, 1, 1]])
```

Define the shorten table.

```
In [32]: text_sub_table = [[[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]]]

text_sub_table = np.array(text_sub_table)
```

In [33]:
```
text_B_sub = []
text_str = ''

for i in range(len(text_B_xor)):
    row = (text_B_xor[i][0] << 1) + (text_B_xor[i][5])
    col = (text_B_xor[i][1] << 3) + (text_B_xor[i][2] << 2) + \
          (text_B_xor[i][3] << 1) + text_B_xor[i][4]
    tmp = text_sub_table[i][row][col]
    tmpstr = '0' * (4 - len(bin(tmp).replace('0b', '')))\
             + bin(tmp).replace('0b', '')
    text_str += tmpstr

text_B_sub = np.array(map(lambda x: int(x), text_str))
```

Successfully shorten the right part into 32-length.

```
In [34]: pretty_print(text_B_sub)
```

```
0 0 0 1 1 0 1 1 0 1    1 1 0 0 0 1 0 0 0 1    0 0 0 1 1 1 1 1 0 0
0 0
```

```
In [35]: len(text_B_sub)
```

```
Out[35]: 32
```

Define the table.

```
In [36]: text_hash_table = [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]
         text_hash_table = np.array(text_hash_table) - 1
```

Displace.

```
In [37]: text_B_hashed = np.array(map(lambda x: text_B_sub[x], text_hash_table))
```

```
In [38]: pretty_print(text_B_hashed)
```

```
1 1 1 0 0 1 1 0 0 0    0 1 1 0 0 1 0 1 1 0    0 1 0 0 0 0 0 0 0 1
1 1
```

XOR with the right part and swap.

```
In [39]: text_A, text_B = text_B, text_A ^ text_B_hashed
```

```
In [40]: pretty_print(text_A)
         print '\n---'
         pretty_print(text_B)
```

```
0 0 0 0 0 0 0 0 1 1    1 1 1 1 1 1 0 0 0 0    0 0 0 0 1 0 0 0 1 0
0 1
---
1 1 1 0 0 1 1 0 1 1    1 0 0 1 1 0 0 1 1 0    1 1 0 0 0 0 1 1 0 0
1 1
```

There will be 15 more loops, remember to change the key in each loop. Besides, in the final loop there is no need to swap left-part and right-part.

```
In [41]: for i in range(1, 16):
             global text_A, text_B
             text_B_extend = np.array(map(lambda x: text_B[x], text_extend_table))
             text_B_xor = text_B_extend ^ keys[i]
             text_B_xor = text_B_xor.reshape(8, 6)
```

```
            text_B_sub = []
            text_str = ''

            for j in range(len(text_B_xor)):
                row = (text_B_xor[j][0] << 1) + (text_B_xor[j][5])
                col = (text_B_xor[j][1] << 3) + (text_B_xor[j][2] << 2) + \
                      (text_B_xor[j][3] << 1) + text_B_xor[j][4]
                tmp = text_sub_table[j][row][col]
                tmpstr = '0' * (4 - len(bin(tmp).replace('0b', ''))) \
                         + bin(tmp).replace('0b', '')
                text_str += tmpstr

            text_B_sub = np.array(map(lambda x: int(x), text_str))
            text_B_hashed = np.array(map(lambda x:text_B_sub[x], text_hash_table))

            if i != 15:
                text_A, text_B = text_B, text_A ^ text_B_hashed
            else:
                text_A = text_A ^ text_B_hashed
```

3.1.4 IP-reverse Function

Put two parts together.

```
In [42]: ciphertext_IPR = np.concatenate((text_A, text_B))
```

Define the IP-reverse function table.

```
In [43]: IPR_table = [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]
         IPR_table = np.array(IPR_table) - 1

In [44]: ciphertext = np.array(map(lambda x: ciphertext_IPR[x], IPR_table))

In [45]: pretty_print(ciphertext)

1 1 0 1 0 1 0 1 0 1    1 0 0 0 1 1 0 0 0 1    0 0 0 0 1 0 1 1 0 0
1 0 1 1 0 1 0 0 1 0    0 1 0 1 1 0 0 1 1 1    0 0 0 1 1 1 1 0 0 1
1 0 0 0
```

3.2 Decryption

Because of the delicate structure of Data Encryption Standard, decryption is just the same as encryption except the keys should be used from `key[16]` to `key[1]`. I attached the python source code `DES_encryption.py` in the zip files, which can encrypt a number of any length. And `DES_decryption.py` take key and the binary string(the output of `DES_encryption.py`) as arguments and extract the plaintext.

```
In [ ]: $ python DES_encryption.py [key] [plaintext]

In [ ]: $ python DES_encryption.py [key] [binaryoutput]
```