Homework Number 02

Name: Yuan Liu

ECN Login: liu1827

Due Date: Jan.30 2020

Problem 1

1.1 The encrypted text

605c6f3e13083a378764e40a8f2254f45b6ca29b034a7780ca45d40d67cc02bd44db1d8e453ceda5
5d9e5465152afef9caeb8ec0f02d82bc7ffabfe89b887e4d60e21e9c9eecc280b91b4f7005743f09ca2
5bad6b3d5208d5f20dea2715d10dec7d59e19e835d9edc78cb6086a7de91ca60d5fa49e79e8550b5
19e0b275243c311346f917df2aff2c18680db97de8f64781405c1c6d57594ced10cec5c5f25533f96
066cf395136779ad02c46a68de8866dc905616d46729cf82d3e402b7daf98adaa11e2bfa27785b77
4487e0d51205b74361d8330187dd32a0d498c4743d023cbb6c8d18eae20dd1dc3ceb1c7477855d4
39e250bc8dc6fbb0c5af42ce813e47b8e0daf5cbafa003e6609bf6ae29030381a819ee5dec49be9bb0
ca9dafde688038f76f9ce344abbc269281db6417db0c423e86aded601870c60fb93e1624b5b8d94df
99ab41f61ca6e846f836a3e1261fcaa2febe41fc17c459b83582f182ff9a65126a7a0dc7e2776aa23c2
0792057edbb681ed4e0e56c9e91cf9fc1b9b1266d66bc30f968978041822c9b9c8ab919429881422
f3c1556b2a16facdabb84677c9aadae181d83aeb66688ffb35dd0dd14dbcabff8b9990375fea81b347
d7318808a2f7231bcf90363a94c2e4a0a9133477336634c7a44e213b2c6e86258509d6770ea58895
bcf57f9e5ad412374897bd67d467d34fd077db85489bd996efcf0352af07645b4614c7a41126731e4
e8357f6c1a9f19d164683c84c9154bb6271438cf1ac748653d1c5f77bc336e2831084b453ff68ac7c
5870ea1f2994058b81e10f5699586c9718df402a1c6cc710a9a0591043525df23249aaa3e9d599cb9
a055ef7bfc360bc19a4baa9ec5f6c2117916669fab00c240e64ce100345f92618cf1b6f16f7b76614d
c26a70de7dea0f37426211591095b172cd327446424512353fb1960c67bfa5fe5cb543d7440cdfdf1
c92ebd7a6e4a14c7c9aadae181d83ae367c2d09fa57949ad3b80db0426c72a576239f51083965ca6
770ea58895bcf571ab1c366b6e2904911554eb2b508534f41b423a02c41c30f100fe12f65fcf3401fd
f2946d24e651446bfab2c48e5bf20b7e8431f1740031213b3aebc2cc8059f4dc37efaa9b16d065f65
3c52ef06ee72fb61a8375b77209ac2236d345892a4aac212665ea415844122f22d777e02aa52e18d
4416d7c33df7eaf44d6d3cc43388b7bb16d8dd25bcfc78b5a5d82ec5657c5d6ff4025aef08b0285aa
47b24eeb850f5dd6e02f1d3fe73a960cd5afdcee7ac882ec8590551c3c016c3c51e9c9a6e7e045fddd
184aa60ba16343fab24601f9b882ec8590551c3c016c3c51e9c9a6e7ef8afba5eb270d8493b506939
fb6f39170b22bf3dbe7f7e55297dc61b9ec15b07abba294bbd23834802469307f609c92325296224
88901efd608835825777cf05527faafff91f6550ea299e9c005501361600c17b99e8d5134523fee0dd
15b65cc157b0b48c6e166023ff42df2446af74f8d28d235d94ba8fd25d09d33972eee1714a2e4a8e5
4310f9f14c918f60c717536a64cca35c181e82dff5a431d60ad981b5f587b7b321527a5014fd5e8de
2f04d713549f570efa46

1.2 The decrypted text

Earlier this week, security researchers took note of a series of changes Linux and Windows developers began rolling out in beta updates to address a critical security flaw: A bug in Intel chips allows low-privilege processes to access memory in the computer's kernel, the machine's most privileged inner sanctum. Theoretical attacks that exploit that bug, based on quirks in features Intel has implemented for faster processing, could allow malicious software to spy deeply into other processes and data on the target computer or smartphone. And on multi-user machines, like the servers run by Google Cloud Services or Amazon Web Services, they could even allow hackers to break out of one user's process, and instead snoop on other processes running on the same shared server. On Wednesday evening, a large team of researchers at Google's Project Zero, universities including the Graz University of Technology, the University of Pennsylvania, the University of Adelaide in Australia, and security companies including Cyberus and Rambus together released the full details of two attacks based on that flaw, which they call Meltdown and Spectre.

1.3 Explanation of the code

By using the provided function “substitution”, “extract_round_key” and “get_encrpytion_key” from prof. kak’s lecture notes, combining with the algorithm provided in the lecture notes, I finished encryption function and save the encrypted text in hexstring by using the attribute “get_hex_string_from_bitvector()”. The decryption function first reads the contents in “encrypted.txt” in hexstring and saves the contents in a bitvector. The next step is to divide the bitvector in blocksize of 64 and put each division in the same algorithm as encryption with the reversed round keys.

1.4 Code for DES_text.py

```
#!/usr/bin/env python3

# Homework Number: 02
# Name: Yuan Liu
# ECN login: liu1827
# Due Date: 1/30/2020

import sys
from BitVector import *
BLOCKSIZE = 64
expansion_permutation = [31, 0, 1, 2, 3, 4,
                         3, 4, 5, 6, 7, 8,
                         7, 8, 9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
                         19, 20, 21, 22, 23, 24,
                         23, 24, 25, 26, 27, 28,
                         27, 28, 29, 30, 31, 0]
```

```
key_permutation_1 = [56, 48, 40, 32, 24, 16, 8, 0, 57, 49, 41, 33, 25, 17,
                     9, 1, 58, 50, 42, 34, 26, 18, 10, 2, 59, 51, 43, 35,
                     62, 54, 46, 38, 30, 22, 14, 6, 61, 53, 45, 37, 29, 21,
                     13, 5, 60, 52, 44, 36, 28, 20, 12, 4, 27, 19, 11, 3]

key_permutation_2 = [13, 16, 10, 23, 0, 4, 2, 27, 14, 5, 20, 9, 22, 18, 11,
                      3, 25, 7, 15, 6, 26, 19, 12, 1, 40, 51, 30, 36, 46,
                     54, 29, 39, 50, 44, 32, 47, 43, 48, 38, 55, 33, 52,
                     45, 41, 49, 35, 28, 31]

shifts_for_round_key_gen = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]


s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [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] ]

s_boxes[1] = [ [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] ]

s_boxes[2] = [ [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] ]

s_boxes[3] = [ [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] ]

s_boxes[4] = [ [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] ]

s_boxes[5] = [ [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] ]

s_boxes[6] = [ [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] ]

s_boxes[7] = [ [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] ]
```

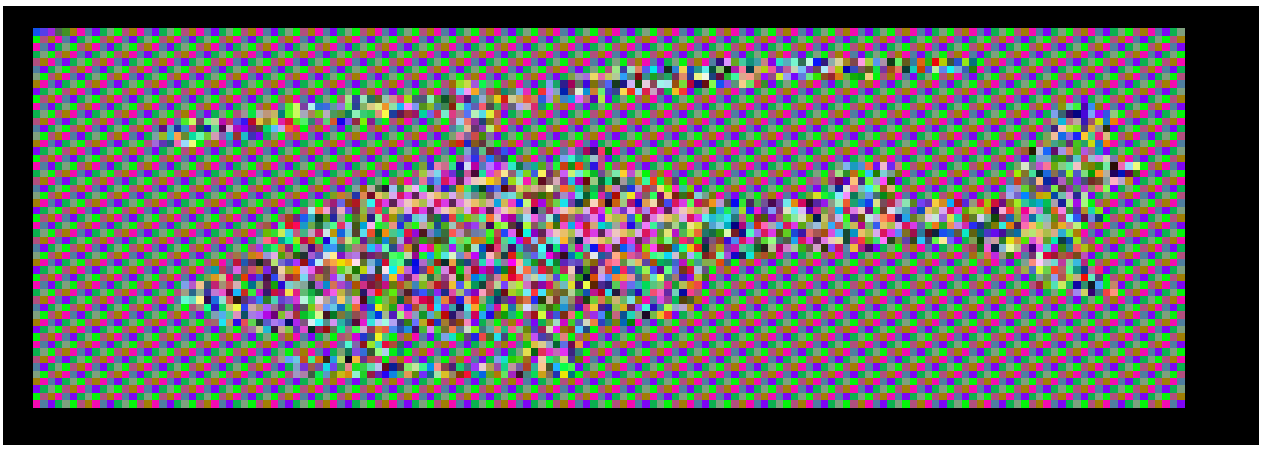
```
p_box_permutation = [15, 6, 19, 20, 28, 11, 27, 16,
                     0, 14, 22, 25, 4, 17, 30, 9,
                     1, 7, 23, 13, 31, 26, 2, 8,
                     18, 12, 29, 5, 21, 10, 3, 24]

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector(intVal =
s_boxes[sindex][row][column], size = 4)
    return output


def extract_round_keys(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys


def get_encryption_key():
    key = open('key.txt', 'r').readline()
    key = BitVector(textstring=key)
    key = key.permute(key_permutation_1)
    return key


def encrypt(input_file, output_file):
    key = get_encryption_key()
    round_keys = extract_round_keys(key)
    bv = BitVector(filename=input_file)
    while bv.more_to_read:
        bitvec = bv.read_bits_from_file(64)
        if bitvec.length() > 0:
            if bitvec.length() < 64:
                bitvec.pad_from_right(64 - bitvec.length())
            [LE, RE] = bitvec.divide_into_two()
            for round_key in round_keys:
                # Expansion Permutation to 48 bits
                newRE = RE.permute(expansion_permutation)
```

```
                # Xoring with the round key
                out_xor = newRE ^ round_key
                # Subsitution with the S_box
                s_box = substitute(out_xor)
                # Permutation with P_box
                p_box = s_box.permute(p_box_permutation)
                REmodified = p_box ^ LE
                LE = RE
                RE = REmodified
            final_string = RE + LE
            with open(output_file, 'a') as fp:
                fp.write(final_string.get_hex_string_from_bitvector())


def decrypt(input_file, output_file):
    key = get_encryption_key()
    round_keys = extract_round_keys(key)
    # Reading from encrpyted hex file
    FILEIN = open(input_file)
    bv = BitVector(hexstring=FILEIN.read())
    if len(bv) % 64:
        bv.pad_from_right(64 - len(bv) % 64)
    for i in range(0, len(bv) // BLOCKSIZE):
        bitvec = bv[i * BLOCKSIZE:(i + 1) * BLOCKSIZE]
        if len(bitvec) > 0:
            [LE, RE] = bitvec.divide_into_two()
            for round_key in round_keys[::-1]:
                # Expansion Permutation to 48 bits
                newRE = RE.permute(expansion_permutation)
                # Xoring with the round key
                out_xor = newRE ^ round_key
                # Subsitution with the S_box
                s_box = substitute(out_xor)
                # Permutation with P_box
                p_box = s_box.permute(p_box_permutation)
                REmodified = p_box ^ LE
                LE = RE
                RE = REmodified
            final_string = RE + LE
            with open(output_file, 'ab') as fp:
                final_string.write_to_file(fp)
            '''
            now comes the hard part --- the substition boxes

            Let's say after the substitution boxes and another
            permutation (P in Section 3.3.4), the output for RE is
            RE_modified.

            When you join the two halves of the bit string
            again, the rule to follow (from Fig. 4 in page 21) is
            either

            final_string = RE followed by (RE_modified xored with LE)

            or
```

```
            final_string = LE followed by (LE_modified xored with RE)

            depending upon whether you prefer to do the substitutions
            in the right half (as shown in Fig. 4) or in the left
            half.

            The important thing to note is that the swap between the
            two halves shown in Fig. 4 is essential to the working
            of the algorithm even in a single-round implementation
            of the cipher, especially if you want to use the same
            algorithm for both encryption and decryption (see Fig.
            3 page 15). The two rules shown above include this swap.
            '''


if __name__ == '__main__':
    if sys.argv[1] == '-e':
        encrypt(sys.argv[2], sys.argv[-1])
    elif sys.argv[1] == '-d':
        decrypt(sys.argv[2], sys.argv[-1])
```

Problem 2

2.1 Encrypted Figure



2.2 Explanation

The encryption algorithm of DES_image.py is the same as the one in DES_text.py. The difference is I write the 3-line header directly from original file to encrypted file.

2.3 Code for DES_image.py

```
#!/usr/bin/env python3

# Homework Number: 02
# Name: Yuan Liu
# ECN login: liu1827
# Due Date: 1/30/2020

import sys
from BitVector import *

expansion_permutation = [31, 0, 1, 2, 3, 4,
                         3, 4, 5, 6, 7, 8,
                         7, 8, 9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
                         19, 20, 21, 22, 23, 24,
                         23, 24, 25, 26, 27, 28,
                         27, 28, 29, 30, 31, 0]

key_permutation_1 = [56, 48, 40, 32, 24, 16, 8, 0, 57, 49, 41, 33, 25, 17,
                     9, 1, 58, 50, 42, 34, 26, 18, 10, 2, 59, 51, 43, 35,
                     62, 54, 46, 38, 30, 22, 14, 6, 61, 53, 45, 37, 29, 21,
                     13, 5, 60, 52, 44, 36, 28, 20, 12, 4, 27, 19, 11, 3]

key_permutation_2 = [13, 16, 10, 23, 0, 4, 2, 27, 14, 5, 20, 9, 22, 18, 11,
                      3, 25, 7, 15, 6, 26, 19, 12, 1, 40, 51, 30, 36, 46,
                     54, 29, 39, 50, 44, 32, 47, 43, 48, 38, 55, 33, 52,
                     45, 41, 49, 35, 28, 31]

shifts_for_round_key_gen = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]


s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [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] ]

s_boxes[1] = [ [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] ]

s_boxes[2] = [ [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] ]

s_boxes[3] = [ [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] ]

s_boxes[4] = [ [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] ]

s_boxes[5] = [ [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] ]

s_boxes[6] = [ [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] ]

s_boxes[7] = [ [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] ]


p_box_permutation = [15, 6, 19, 20, 28, 11, 27, 16,
                     0, 14, 22, 25, 4, 17, 30, 9,
                     1, 7, 23, 13, 31, 26, 2, 8,
                     18, 12, 29, 5, 21, 10, 3, 24]

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector(intVal =
s_boxes[sindex][row][column], size = 4)
    return output


def extract_round_keys(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
```

```
    return round_keys


def get_encryption_key():
    key = open('key.txt', 'r').readline()
    key = BitVector(textstring=key)
    key = key.permute(key_permutation_1)
    return key


def encrypt(input_file, output_file):
    key = get_encryption_key()
    round_keys = extract_round_keys(key)
    bv = BitVector(filename=input_file)

    #Save the header in a list
    msg = []
    with open(input_file, "rb") as FILEIN:
        msg = (FILEIN.readlines()[0:3])

    #Write the header to the output file
    with open(output_file, "wb") as FILEOUT:
        for ele in msg:
            FILEOUT.write(ele)

    #Move the reading pointer to correnct position
    bv.read_bits_from_file(len(msg)*8)
    while bv.more_to_read:
        bitvec = bv.read_bits_from_file(64)
        if bitvec.length() > 0:
            if bitvec.length() < 64:
                bitvec.pad_from_right(64 - bitvec.length())
            [LE, RE] = bitvec.divide_into_two()
            for round_key in round_keys:
                # Expansion Permutation to 48 bits
                newRE = RE.permute(expansion_permutation)
                # Xoring with the round key
                out_xor = newRE ^ round_key
                # Subsitution with the S_box
                s_box = substitute(out_xor)
                # Permutation with P_box
                p_box = s_box.permute(p_box_permutation)
                REmodified = p_box ^ LE
                LE = RE
                RE = REmodified
            final_string = RE + LE
            with open(output_file, 'ab') as fp:
                final_string.write_to_file(fp)


if __name__ == '__main__':
    encrypt(sys.argv[1], sys.argv[-1])
```