Homework Number 02

Name: Yuan Liu

ECN Login: liu1827

Due Date: Jan.30 2020

Problem 1

1.1 The encrypted text



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])