Homework Number: 2  
Name: Jose Luis Tejada  
ECN Login: tejada  
Due Date: Thursday 1/30/2020 at 4:29PM

***Problem 1:***

**Code:**

*#!/usr/bin/env python3***import** sys  
**from** BitVector **import** \*  
  
*#Permutation / Substitution arrays*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]  
  
  
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]  
  
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] ]  
  
pbox\_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** encrypt(message, key, encrypted):  
 *#Obtain key and round keys* encryption\_key, round\_keys = generate\_keys(key)  
  
 *#Convert Message to bitvector* message\_bv = BitVector(filename=message) *#Assume conversion with textstring.  
  
 #Continue for entirety of message* encrypted\_bv = BitVector(size=0)  
 **while**(message\_bv.more\_to\_read):  
 *#Obtain Segment* message\_bv\_segment = message\_bv.read\_bits\_from\_file(64)  
  
 *# Pad read bits with 0's when block size is not 64* **if** message\_bv\_segment.size < 64:  
 message\_bv\_segment.pad\_from\_right(64-len(message\_bv\_segment))  
  
 *#Feistel function* [Left, Right] = message\_bv\_segment.divide\_into\_two() *# Perform left/right division* **for** r\_key **in** round\_keys: *#For each segement of 64-bits perform 16 rounds of encryption* new\_Right = Right.deep\_copy()  
 new\_Right = new\_Right.permute(permute\_list=expansion\_permutation) *#Perform expansion permutation on right half to 48 bits* new\_Right ^= r\_key *# Perform xor with key, must be 48bits  
  
 # Perform substitution with s-boxes, remember input -> 48bits, output -> 32bits* s\_box\_output = substitution(new\_Right)  
  
 p\_box\_Right = s\_box\_output.permute(permute\_list=pbox\_permutation) *# Perform permutation with P Box* p\_box\_Right ^= Left *# Perform final Xoring with requisite half  
  
 #Set left and right for next round.* Left = Right.deep\_copy()  
 Right = p\_box\_Right  
 *#End of 16 Encryption Rounds for current segment  
 #Append each segment's left and right halves together.* encrypted\_bv += (Right + Left) *#Check Method of usage  
  
 #Once finished reading, write bitvector to encrypted\_ppm* encrypted\_fp = open(file=encrypted, mode=**'w'**)  
 encrypted\_bv\_hex\_string = encrypted\_bv.get\_hex\_string\_from\_bitvector()  
 encrypted\_fp.write(encrypted\_bv\_hex\_string)  
  
 *#Close files* encrypted\_fp.close()  
  
 **return  
  
def** decrypt(encrypted,key, decrypted):  
 *# Obtain key and round keys* encryption\_key, round\_keys = generate\_keys(key)  
 *#Attempt to flip round keys so that key 16 is run 1st* round\_keys.reverse()  
  
 *# Convert Message to bitvector* fp = open(file=encrypted, mode=**'r'**)  
 encrypted\_bv = BitVector(hexstring=fp.read()) *# Assume conversion from hexstring to bitvector  
  
 # Continue for entirety of message* decrypted\_bv = BitVector(size=0)  
  
 *#Split hexfile into 64 bit chunks* encrypted\_bv\_split = [encrypted\_bv[i\*64:(i+1)\*64] **for** i **in** range((len(encrypted\_bv) // 64))] *#Check if correct* **for** encrypted\_bv\_segment **in** encrypted\_bv\_split:  
 *# Obtain Segments* [Left, Right] = encrypted\_bv\_segment.divide\_into\_two() *# Perform left/right division* **for** r\_key **in** round\_keys: *# For each segement of 64-bits perform 16 rounds of encryption* new\_Right = Right.deep\_copy()  
 new\_Right = new\_Right.permute(permute\_list=expansion\_permutation) *# Perform expansion permutation on right half to 48 bits* new\_Right ^= r\_key *# Perform xor with key, must be 48bits  
  
 # Perform substitution with s-boxes, remember input -> 48bits, output -> 32bits* s\_box\_output = substitution(new\_Right)  
  
 p\_box\_Right = s\_box\_output.permute(permute\_list=pbox\_permutation) *# Perform permutation with P Box* p\_box\_Right ^= Left *# Perform final Xoring with requisite half* Left = Right.deep\_copy()  
 Right = p\_box\_Right  
 *# End of 16 Encryption Rounds for current segment  
 # Append each segment's left and right halves together.* decrypted\_bv += (Right + Left) *# Check Method of usage  
  
 # Once finished reading, write bitvector to decrypted* decrypted\_fp = open(file=decrypted, mode=**'w'**)  
 decrypted\_bv\_text = decrypted\_bv.get\_text\_from\_bitvector()  
 decrypted\_fp.write(decrypted\_bv\_text)  
  
 *#Close files* fp.close()  
 decrypted\_fp.close()  
  
 **return  
  
def** generate\_keys(key):  
 *#Obtain Encryption key* key\_file\_pointer = open(file=key, mode=**'r'**)  
 key\_string = key\_file\_pointer.read()  
 key\_bv = BitVector(textstring=key\_string)  
 key\_bv\_p = key\_bv.permute(permute\_list=key\_permutation\_1)  
  
 *#Generate Round Keys* round\_keys = []  
 encryption\_key\_bv = key\_bv\_p.deep\_copy()  
 **for** round\_num **in** range(16):  
 [left\_key, right\_key] = encryption\_key\_bv.divide\_into\_two()  
 round\_shift = shifts\_for\_round\_key\_gen[round\_num]  
 left\_key << round\_shift  
 right\_key << round\_shift  
 complete\_key = left\_key + right\_key  
 round\_key = complete\_key.permute(key\_permutation\_2)  
 round\_keys.append(round\_key)  
  
 **return** key\_bv\_p, round\_keys  
  
**def** substitution(half\_block\_48):  
  
 s\_box\_output = BitVector(size=32)  
 s\_segments = [half\_block\_48[(i \* 6):(i \* 6 + 6)] **for** i **in** range(8)] *# Determine segments for substitution* **for** s\_index **in** range(len(s\_segments)): *#For each segment* row = 2 \* s\_segments[s\_index][0] + s\_segments[s\_index][-1] *# Determine row of substitution* col = int(s\_segments[s\_index][1:-1]) *# Determine column in row of s\_box to be substituted with* s\_box\_output[s\_index \* 4:(s\_index \* 4 + 4)] = BitVector(intVal=s\_boxes[s\_index][row][col],size=4) *# Perform substitution* **return** s\_box\_output  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 *# Assume correct number of arguments and format* **if** len(sys.argv) != 5:  
 print(**"Incorrect number of Arguments"**)  
 exit(1)  
 **if** sys.argv[1] == **'-e'**:  
 message = sys.argv[2]  
 key = sys.argv[3]  
 encrypted = sys.argv[4]  
 encrypt(message, key, encrypted)  
 **elif** sys.argv[1] == **'-d'**:  
 encrypted = sys.argv[2]  
 key = sys.argv[3]  
 decrypted = sys.argv[4]  
 decrypt(encrypted,key, decrypted)  
 **else**:  
 print(**"Incorrect Encryption/Decryption Option."**)  
 exit(1)

**Encrypted Plaintext:**

**Decrypted Plaintext:**

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.

**Explanation:**

In this problem the goal was to implement the Data Encryption Standard algorithm and be able to both encrypt and decrypt a text file with a given key in a text file. We first read the 8-character key and perform a permutation with the key permutation 1 array, resulting in a new 48-bit key. We use this key to generate all 16 round keys to later be used to generate the encrypted file. The encryption key is then divided into two portions, and depending on the round number, we perform a set number of left-shifting operations. We then join both halves back together and apply a second permutation on this key and append it to a list. Having this, we can convert the plaintext into a bit vector and reading in 64 bit segments one at a time padding the text appropriately if it is less than 64 bits long and divide each segment into two 32 bit segments. For each round key we make a copy of the right portion and permute it using the expansion permutation to obtain a 48-bit segment and xor it with the current round key. We then proceed to obtain eight 6 bit segments and perform a substitution on the middle 4 bits of each using the s\_boxes arrays and two outer bits as indices, obtaining a 32 bit segment as output. Finally, we perform an additional permutation and XOR this segment with the original left segment, finally, we swap right a left segments and repeat the whole process for all keys and for all 64 bit segments in the file. We can then finally write the encrypted file. For decryption the process is exactly the same as for decryption.

***Problem 2:***

**Code:**

**import** sys  
**from** BitVector **import** \*  
  
*#Permutation / Substitution arrays*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]  
  
  
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]  
  
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] ]  
  
pbox\_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** encrypt(input\_image, key, encrypted\_ppm):  
 *#Open input\_image, Read three lines as header* image\_fp = open(file=input\_image, mode=**'rb'**)  
 image\_header = []  
 image\_header.append(image\_fp.readline())  
 image\_header.append(image\_fp.readline())  
 image\_header.append(image\_fp.readline())  
  
 *#Read remaining lines as data to encrypt.* image\_data = image\_fp.read()  
 image\_bv = BitVector(rawbytes=image\_data)*#Convert input\_image data to bitvector  
  
 # Close files* image\_fp.close()  
  
 *# Obtain key and round keys* encryption\_key, round\_keys = generate\_keys(key)  
  
 *# Split contents of input\_image, must pad last byte if not divisible by 64* num\_segments = len(image\_bv) // 64  
 image\_bv\_split = [image\_bv[i\*64:(i+1)\*64] **for** i **in** range(num\_segments)]  
 **if** (len(image\_bv) % 64 != 0):  
 last\_segment = image\_bv[num\_segments\*64:len(image\_bv)]  
 last\_segment.pad\_from\_right(64 - (len(image\_bv) - num\_segments\*64))  
 image\_bv\_split.append(last\_segment)  
  
 *# Write encrypted\_ppm bitvector to ppm  
 # Use 'wb' as write binary option* encrypted\_fp = open(file=encrypted\_ppm, mode=**'wb'**)  
 **for** h **in** image\_header:  
 encrypted\_fp.write(h)  
  
  
 *#Continue for entirety of message* encrypted\_bv = BitVector(size=0)  
 **for** image\_bv\_segment **in** image\_bv\_split:  
 [Left, Right] = image\_bv\_segment.divide\_into\_two()  
 **for** r\_key **in** round\_keys: *# For each segement of 64-bits perform 16 rounds of encryption* new\_Right = Right.deep\_copy()  
 new\_Right = new\_Right.permute(permute\_list=expansion\_permutation) *# Perform expansion permutation on right half to 48 bits* new\_Right ^= r\_key *# Perform xor with key, must be 48bits  
  
 # Perform substitution with s-boxes, remember input -> 48bits, output -> 32bits* s\_box\_output = substitution(new\_Right)  
  
 p\_box\_Right = s\_box\_output.permute(permute\_list=pbox\_permutation) *# Perform permutation with P Box* p\_box\_Right ^= Left *# Perform final Xoring with requisite half* Left = Right.deep\_copy()  
 Right = p\_box\_Right  
 *# End of 16 Encryption Rounds for current segment  
 # Append each segment's left and right halves together.* encrypted\_bv = (Right + Left) *# Check Method of usage* encrypted\_bv.write\_to\_file(file\_out=encrypted\_fp)  
  
 *#Close File* encrypted\_fp.close()  
  
 **return  
  
  
def** generate\_keys(key):  
 *#Obtain Encryption key* key\_file\_pointer = open(file=key, mode=**'r'**)  
 key\_string = key\_file\_pointer.read()  
 key\_bv = BitVector(textstring=key\_string)  
 key\_bv\_p = key\_bv.permute(permute\_list=key\_permutation\_1)  
  
 *#Generate Round Keys* round\_keys = []  
 encryption\_key\_bv = key\_bv\_p.deep\_copy()  
 **for** round\_num **in** range(16):  
 [left\_key, right\_key] = encryption\_key\_bv.divide\_into\_two()  
 round\_shift = shifts\_for\_round\_key\_gen[round\_num]  
 left\_key << round\_shift  
 right\_key << round\_shift  
 complete\_key = left\_key + right\_key  
 round\_key = complete\_key.permute(key\_permutation\_2)  
 round\_keys.append(round\_key)  
  
 **return** key\_bv\_p, round\_keys  
  
**def** substitution(half\_block\_48):  
  
 s\_box\_output = BitVector(size=32)  
 s\_segments = [half\_block\_48[(i \* 6):(i \* 6 + 6)] **for** i **in** range(8)] *# Determine segments for substitution* **for** s\_index **in** range(len(s\_segments)): *#For each segment* row = 2 \* s\_segments[s\_index][0] + s\_segments[s\_index][-1] *# Determine row of substitution* col = int(s\_segments[s\_index][1:-1]) *# Determine column in row of s\_box to be substituted with* s\_box\_output[s\_index \* 4:(s\_index \* 4 + 4)] = BitVector(intVal=s\_boxes[s\_index][row][col],size=4) *# Perform substitution* **return** s\_box\_output  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 **if** len(sys.argv) != 4:  
 print(**"Incorrect Number of Arguements"**)  
 exit(1)  
 image = sys.argv[1]  
 key = sys.argv[2]  
 encrypted = sys.argv[3]  
 encrypt(image, key, encrypted)

**Encrypted Image:**

****

**Explanation:**

In order to encrypt the provided image, there are some additional steps that must be taken so that we can use the same algorithm as with the text encryption. First, we must read the header of the file and image data separately so that we don’t encrypt the header of the image. We can then proceed to write the binary header onto the encrypted file. After doing so we convert the bytes of the image directly into a bit vector and proceed to divide it into 64 bit segments, and padding them appropriately. From this point forward, we can proceed to encrypt the image with the same scheme as with the text encryption in problem 1, with the only caveat being that we write each successfully encrypted 64 bit segment to the file upon completion instead of writing it all at once (For speed reasons).