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

<u>Problem 1:</u>

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:

```
96a1038212ed8e645a7dd8fed46e3dc942d37f063536e948dc2a8ed0193090236cd0ea09995fd353fa9966
1653e9d1899ede589138b0cddcaf7937af724c98b7c08b553f2eda1c499366016406dc0b859c23aca1d513
527340d5bcba0a1d8932d5416291bbd2a467847c563bc3d63a82576a72ebdc3b9b5c1db5b91a6e243d4dfb
7da5ca30614502f71154543e89cbba96788a9332f11f09a1f08de5fd29312700ed8fc037b09e2a3cfe8786
a589211a80a19808c2d86bd3402dda3f6b656c96418b2592fd0ca685177533063d669e2e891d615c8fc859
a2a01e2040b78a61566d8efedfe557f6bae638bc32345f7b5180b956059f4245b4e48468fe6610ca9dd9d5
3f1c4a43ded7720a942a5d6b0c2bba8f5793603930efdd24ccfc1cea78996fa2d9223df8dbbe8acd83b7fc
5b73e4cf935179cd7836fdabc2893edc2f1585006907fe3b96236633b89c58a82ec6696163a61a6f239290
4b6318edca98bea88499e53656b268b9dd196f48d79ea1cd74fc7bb748df3f6e5c9233200dda8477026202
be048d8fcac8206f33c381ee475dd22e92a6fe3d39f8a193b955ab6a37a05c46f4db1a4c5687d16570520d
81fe0e7b7ab3a84d65f74e3d2c813c473d6d01b144db15ef16bb7abf41f25a55579d4bde1f56fd66b92cad
4233efacb4467d3f311b0eae6a8d636f22705cb1ad0891325a5011a59dc89f73e149a457f78abbf2bff1a5
54c8fd04b06277814b8210348538e27aefc452b909e2c555e48ba74e8d588da3ee26774c608c67c0fc3e25
97863e32e0220fbc430ace69319e3331c8d8288f10edc0f75e151d3db6a2d41c0f1c6e7fc6c5f8c1a3e7ec
a2beb5d5602bf8917e17426e69d05b19a124af859ab73de79c443608c287202f5784bd18b17714b3fd6bec
b95bc6c90e3f87fa6b24a42c6c53eab5145cc7c6b2e31528c578fe367dab41c4cf29d58b0a810c95ae6a19
8a7866788c03711ea05c46f4db1a4c56b5dcb299e40ce6cd67fe214e96d1b4ff9a774e0e46fb6d353f311b
0eae6a8d63aa8e97e07ff56ab339e13cf7e4b5b5641797d1d233b0211f05f33590ce58b7c146e2f2588a0a
87f71ea1a643b4faf31478b86806d7995913d5f770643445841048177b01b617066c3a25a77c1ecafbbb99
a3e392c18d2cc9d0d002cf5281d43a7c64a60768ca3269e09b50edf4f7a9a74ba1208a35b7d61517ca4a84
013a057ce47749fda25592edb244c52ec747521980659e973b85cbd17318fb458508e77f154670d05a4118
cc9b12e2d5a0ce3057670c7799e783ad37b559aba8cb38abf07b1cceee65f508f1602a49d7881556bc670c
7799e783ad37b559aba8cb38abf0b4676866bbb0e3f043c4208078dfc99e179cfc066839c345ff1b7d8b81
937097a09351bd60af518898f199e1070e658b2167546a1f884e47df43819a748de22d3c1832f071be306e
c6063fbb15d6fbf7667e091f70cc4b48244bbe2805c7575c54cfb68311fe15b7a14f3fc829c08778c8ce11
99055ce933d6d3adbeb4f8dfae2ef4d0d1bcda813449607872604365559a223e2233dc08b30ff547b25d798
d85eef2cb1f150fbd16902352dcc9b996fd54af4b70531a676fc
```

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:

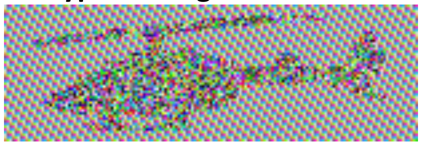
In this problem the goal was to implement the Data Encryption Standard algorithm and be able to both encrypt and decrypt a text file with a given key in a text file. We first read the 8-character key and perform a permutation with the key permutation 1 array, resulting in a new 48-bit key. We use this 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).