Terrence Randall

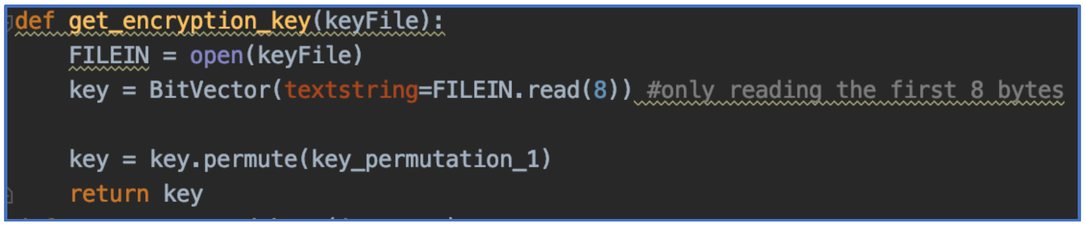
HW02 Summary

Problem # 1:

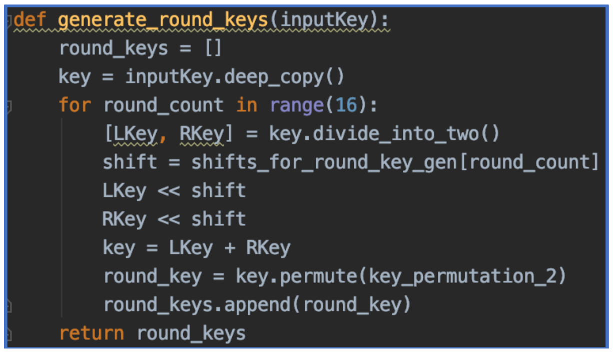
Code:

Note: substitute, get_encryption_key, and generate_round_keys lightly (or not) modified

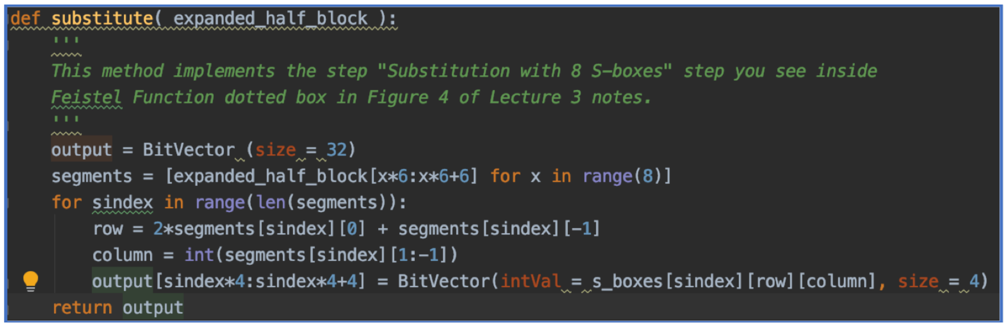
Given Functions:



```python
def get_encryption_key(keyFile):
    FILEIN = open(keyFile)
    key = BitVector(textstring=FILEIN.read(8)) #only reading the first 8 bytes

    key = key.permute(key_permutation_1)
    return key
```



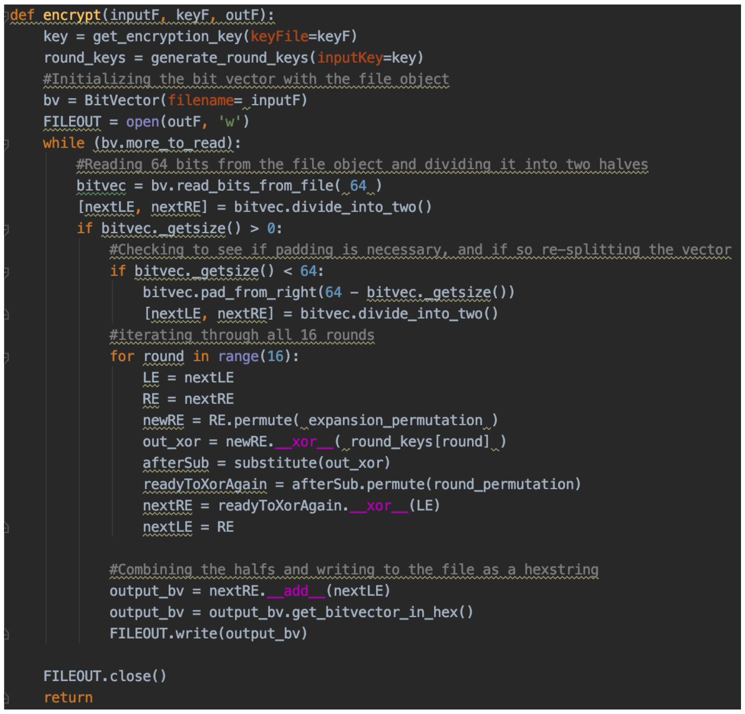
```python
def generate_round_keys(inputKey):
    round_keys = []
    key = inputKey.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
 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



```python
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
```

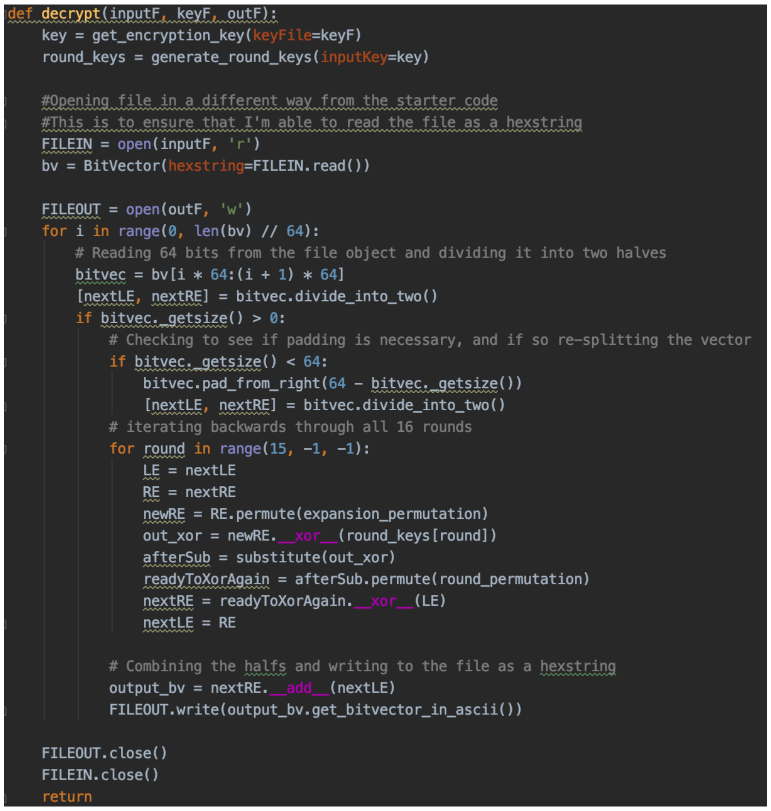
Terrence Randall

Encrypt function:



```python
def encrypt(inputF, keyF, outF):
    key = get_encryption_key(keyFile=keyF)
    round_keys = generate_round_keys(inputKey=key)
    #Initializing the bit vector with the file object
    bv = BitVector(filename= inputF)
    FILEOUT = open(outF, 'w')
    while (bv.more_to_read):
        #Reading 64 bits from the file object and dividing it into two halves
        bitvec = bv.read_bits_from_file( 64 )
        [nextLE, nextRE] = bitvec.divide_into_two()
        if bitvec._getsize() > 0:
            #Checking to see if padding is necessary, and if so re-splitting the vector
            if bitvec._getsize() < 64:
                bitvec.pad_from_right(64 - bitvec._getsize())
                [nextLE, nextRE] = bitvec.divide_into_two()
            #iterating through all 16 rounds
            for round in range(16):
                LE = nextLE
                RE = nextRE
                newRE = RE.permute( expansion_permutation )
                out_xor = newRE.__xor__( round_keys[round] )
                afterSub = substitute(out_xor)
                readyToXorAgain = afterSub.permute(round_permutation)
                nextRE = readyToXorAgain.__xor__(LE)
                nextLE = RE

            #Combining the halfs and writing to the file as a hexstring
            output_bv = nextRE.__add__(nextLE)
            output_bv = output_bv.get_bitvector_in_hex()
            FILEOUT.write(output_bv)

    FILEOUT.close()
    return
```

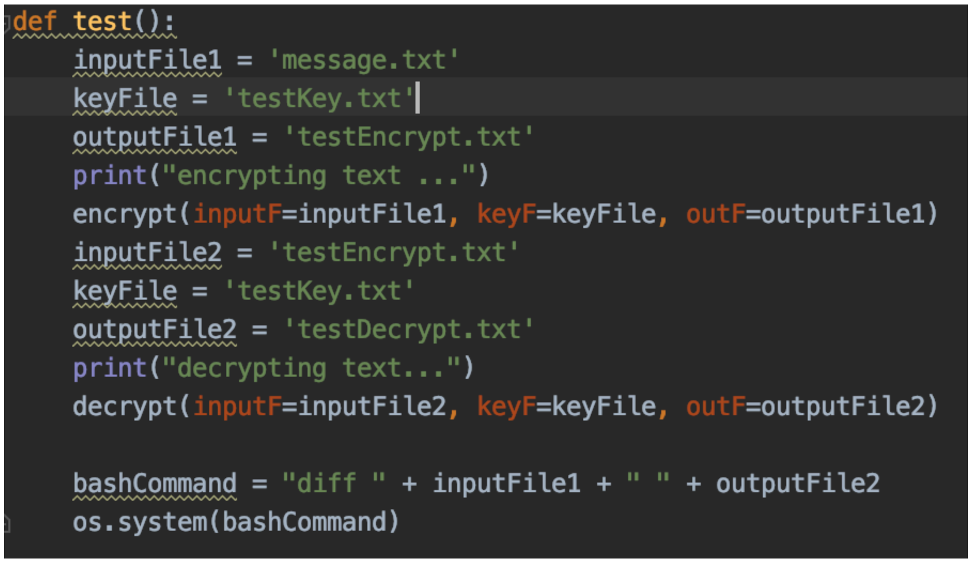
Decrypt function:



```python
def decrypt(inputF, keyF, outF):
    key = get_encryption_key(keyFile=keyF)
    round_keys = generate_round_keys(inputKey=key)

    #Opening file in a different way from the starter code
    #This is to ensure that I'm able to read the file as a hexstring
    FILEIN = open(inputF, 'r')
    bv = BitVector(hexstring=FILEIN.read())

    FILEOUT = open(outF, 'w')
    for i in range(0, len(bv) // 64):
        # Reading 64 bits from the file object and dividing it into two halves
        bitvec = bv[i * 64:(i + 1) * 64]
        [nextLE, nextRE] = bitvec.divide_into_two()
        if bitvec._getsize() > 0:
            # Checking to see if padding is necessary, and if so re-splitting the vector
            if bitvec._getsize() < 64:
                bitvec.pad_from_right(64 - bitvec._getsize())
                [nextLE, nextRE] = bitvec.divide_into_two()
            # iterating backwards through all 16 rounds
            for round in range(15, -1, -1):
                LE = nextLE
                RE = nextRE
                newRE = RE.permute(expansion_permutation)
                out_xor = newRE.__xor__(round_keys[round])
                afterSub = substitute(out_xor)
                readyToXorAgain = afterSub.permute(round_permutation)
                nextRE = readyToXorAgain.__xor__(LE)
                nextLE = RE

            # Combining the halfs and writing to the file as a hexstring
            output_bv = nextRE.__add__(nextLE)
            FILEOUT.write(output_bv.get_bitvector_in_ascii())

    FILEOUT.close()
    FILEIN.close()
    return
```

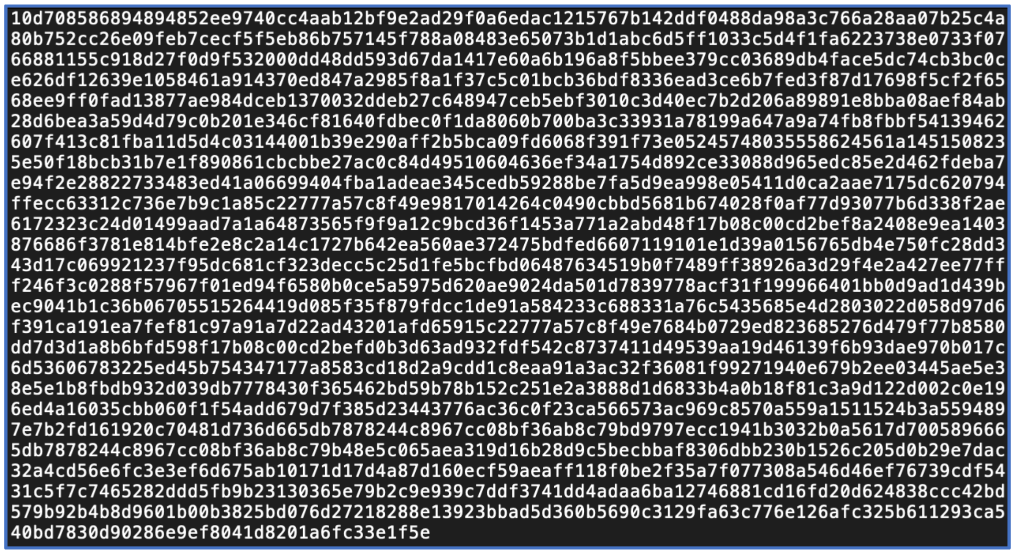
Terrence Randall

Test Code:



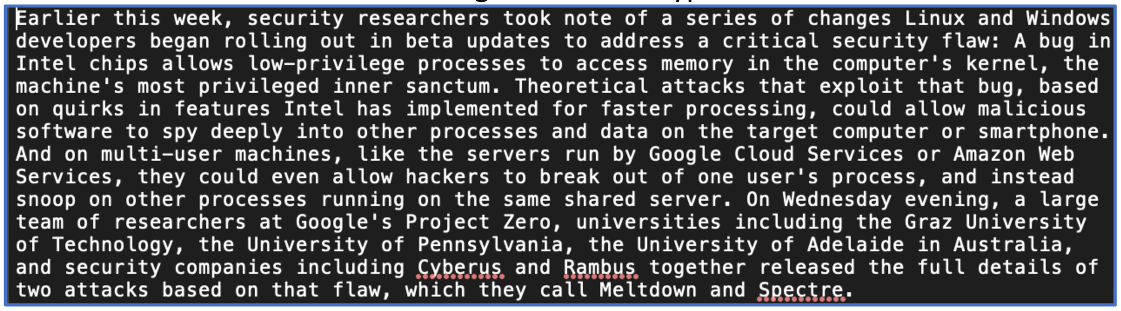
```
def test():
    inputFile1 = 'message.txt'
    keyFile = 'testKey.txt'
    outputFile1 = 'testEncrypt.txt'
    print("encrypting text ...")
    encrypt(inputF=inputFile1, keyF=keyFile, outF=outputFile1)
    inputFile2 = 'testEncrypt.txt'
    keyFile = 'testKey.txt'
    outputFile2 = 'testDecrypt.txt'
    print("decrypting text...")
    decrypt(inputF=inputFile2, keyF=keyFile, outF=outputFile2)

    bashCommand = "diff " + inputFile1 + " " + outputFile2
    os.system(bashCommand)
```

Result: (output from encryption, and decryption)



```
10d708586894894852ee9740cc4aab12bf9e2ad29f0a6edac1215767b142ddf0488da98a3c766a28aa07b25c4a
80b752cc26e09feb7cecf5f5eb86b757145f788a08483e65073b1d1abc6d5ff1033c5d4f1fa6223738e0733f07
66881155c918d27f0d9f532000dd48dd593d67da1417e60a6b196a8f5bbee379cc03689db4face5dc74cb3bc0c
e626df12639e1058461a914370ed847a2985f8a1f37c5c01bcb36bdf8336ead3ce6b7fed3f87d17698f5cf2f65
68ee9ff0fad13877ae984dceb1370032ddeb27c648947ceb5ebf3010c3d40ec7b2d206a89891e8bba08aef84ab
28d6bea3a59d4d79c0b201e346cf81640fdbec0f1da8060b700ba3c33931a78199a647a9a74fb8fbbf54139462
607f413c81fba11d5d4c03144001b39e290aff2b5bca09fd6068f391f73e05245748035558624561a145150823
5e50f18bcb31b7e1f890861cbcbbe27ac0c84d49510604636ef34a1754d892ce33088d965edc85e2d462fdeba7
e94f2e28822733483ed41a06699404fba1adeae345cedb59288be7fa5d9ea998e05411d0ca2aae7175dc620794
ffecc63312c736e7b9c1a85c22777a57c8f49e9817014264c0490cbbd5681b674028f0af77d93077b6d338f2ae
6172323c24d01499aad7a1a64873565f9f9a12c9bcd36f1453a771a2abd48f17b08c00cd2bef8a2408e9ea1403
876686f3781e814bfe2e8c2a14c1727b642ea560ae372475bdfed6607119101e1d39a0156765db4e750fc28dd3
43d17c069921237f95dc681cf323decc5c25d1fe5bcfbd06487634519b0f7489ff38926a3d29f4e2a427ee77ff
f246f3c0288f57967f01ed94f6580b0ce5a5975d620ae9024da501d7839778acf31f199966401bb0d9ad1d439b
ec9041b1c36b06705515264419d085f35f879fdcc1de91a584233c688331a76c5435685e4d2803022d058d97d6
f391ca191ea7fef81c97a91a7d22ad43201afd65915c22777a57c8f49e7684b0729ed823685276d479f77b8580
dd7d3d1a8b6bfd598f17b08c00cd2befd0b3d63ad932fdf542c8737411d49539aa19d46139f6b93dae970b017c
6d53606783225ed45b754347177a8583cd18d2a9cdd1c8eaa91a3ac32f36081f99271940e679b2ee03445ae5e3
8e5e1b8fbdb932d039db7778430f365462bd59b78b152c251e2a3888d1d6833b4a0b18f81c3a9d122d002c0e19
6ed4a16035cbb060f1f54add679d7f385d23443776ac36c0f23ca566573ac969c8570a559a1511524b3a559489
7e7b2fd161920c70481d736d665db7878244c8967cc08bf36ab8c79bd9797ecc1941b3032b0a5617d700589666
5db7878244c8967cc08bf36ab8c79b48e5c065aea319d16b28d9c5becbbaf8306dbb230b1526c205d0b29e7dac
32a4cd56e6fc3e3ef6d675ab10171d17d4a87d160ecf59aeaff118f0be2f35a7f077308a546d46ef76739cdf54
31c5f7c7465282ddd5fb9b23130365e79b2c9e939c7ddf3741dd4adaa6ba12746881cd16fd20d624838ccc42bd
579b92b4b8d9601b00b3825bd076d27218288e13923bbad5d360b5690c3129fa63c776e126afc325b611293ca5
40bd7830d90286e9ef8041d8201a6fc33e1f5e
```

<Hexstring file after encryption>



```
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.
```

<Decrypted file after decrypting the above hexstring file>

Terrence Randall

Code Summary: (explanation)

After ensuring that there are enough input arguments and the second is either “-e” or “-d”, the script will begin by assigning the string of input files to their own variables. Assuming that we begin by encrypting a file, each of these variables will be passed through to the encrypt function where the encryption key and the round keys are found through two different dedicated functions. The “get_encryption_key” function simply reads the key file and permutes the retrieved bit-key so that it returns a 56-bit key. And the “get_round_keys” function takes the 56-bit key as a parameter and does an operation on it (splits, rotates, combines, and permutates) 16 different time to return a list of 16 different round keys.

Next, the plaintext file is opened as a file-object within a bit vector, so that it can be iteratively read in a while-loop until the bit-vector gets to the end of the file. Inside this while-loop, blocks of 64 bits are read at a time. These blocks are checked for being 64 bits (if not they’re padded), then split into a left-half and right-half, and the right half proceeds through a sequence of operations (expansion, XORed w/ round key, substitution, permuted, and XORed w/ the left-half). The result of these operations is then assigned to the next left-half block for the next round, while the unaltered right-half is then assigned to the next right-half block (Note: This process still happens after the last round, but it is immediately reversed). And each block is writte directly to the output file as a hexstring after it is encrypted.

In decryption, the process is the same up until the time of reading from the file. To read from the file I decided to open it with a file pointer to help in facilitating reading the file as a hexstring with the bit-vector object. Because of the previous difference, the decrypting uses a for loop to iterate through the encrypted bit-vector (rather than iterating through the file). Further, the decryption is the same process as the encryption. And when the block is decrypted it it writte to the output file as a text string.

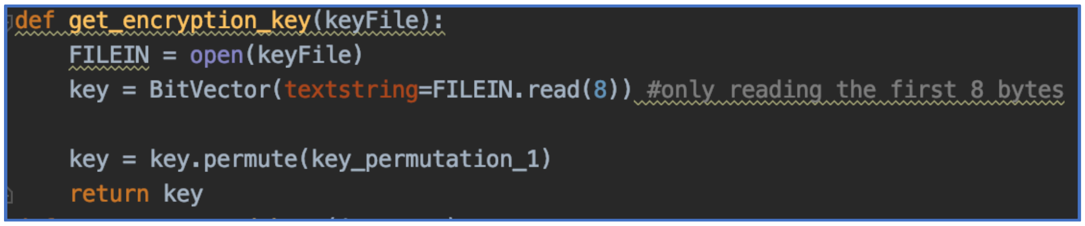
Terrence Randall

Problem # 2:

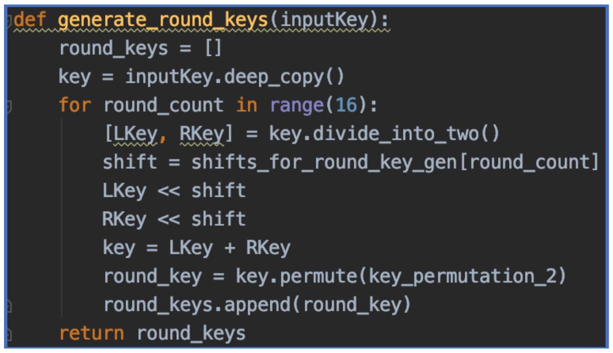
Code:

Note: substitute, get_encryption_key, and generate_round_keys lightly (or not) modified

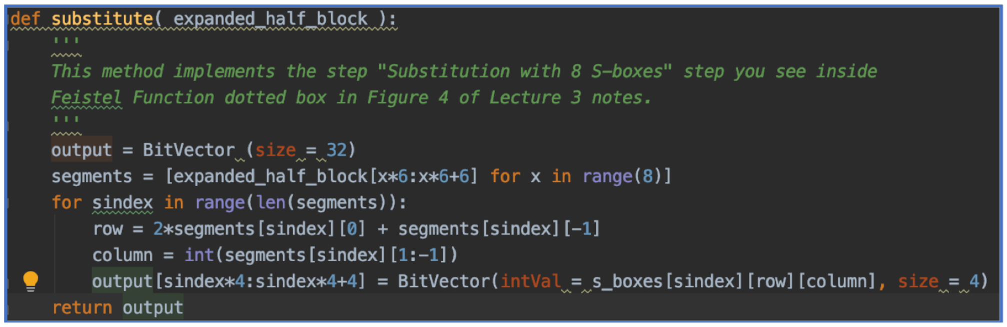
Given Functions:



```python
def get_encryption_key(keyFile):
    FILEIN = open(keyFile)
    key = BitVector(textstring=FILEIN.read(8)) #only reading the first 8 bytes

    key = key.permute(key_permutation_1)
    return key
```



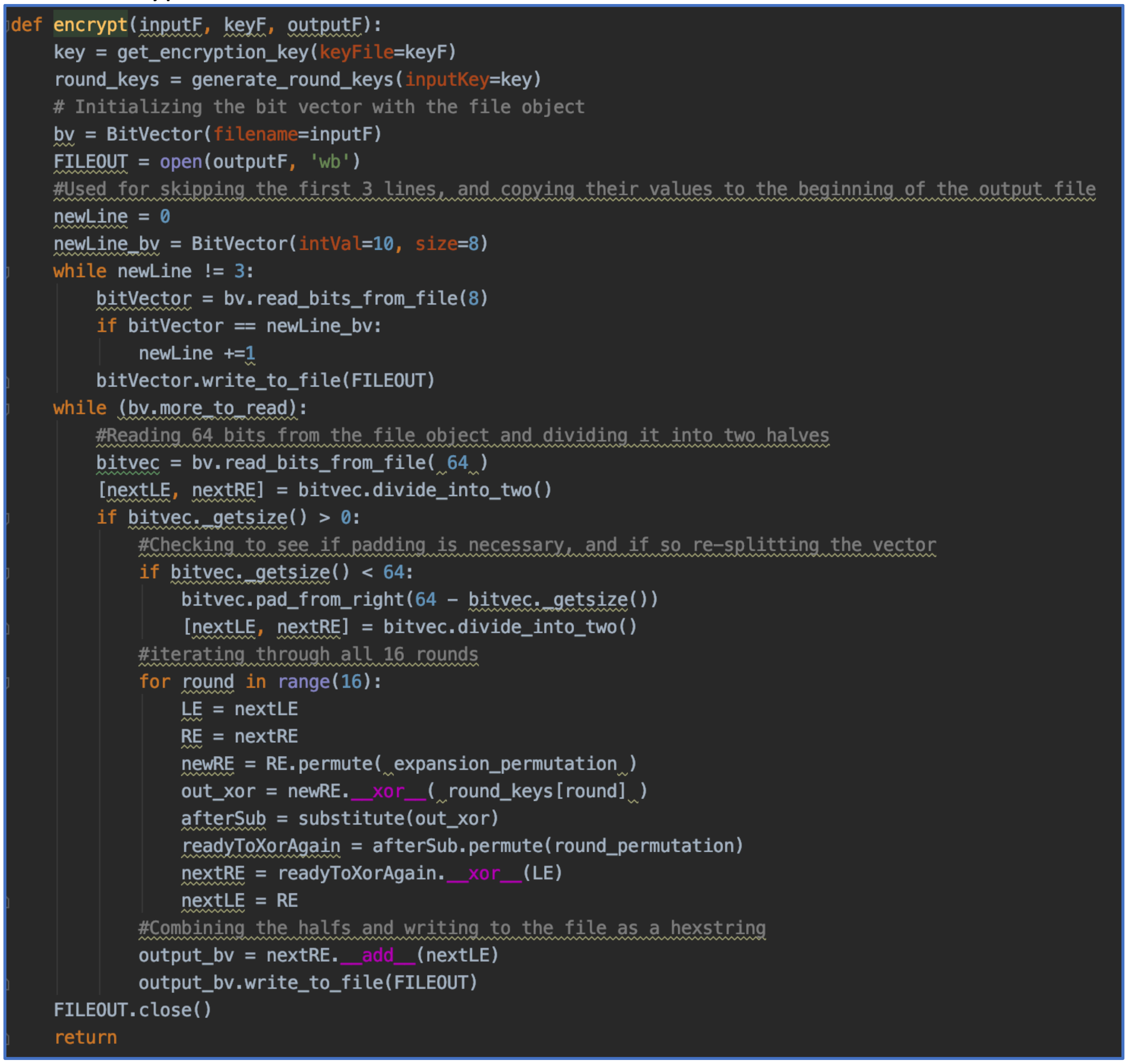
```python
def generate_round_keys(inputKey):
    round_keys = []
    key = inputKey.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
 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



```python
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
```

Terrence Randall

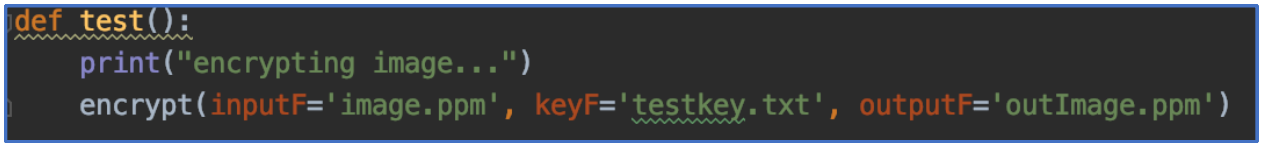
Encrypt function:



```python
def encrypt(inputF, keyF, outputF):
    key = get_encryption_key(keyFile=keyF)
    round_keys = generate_round_keys(inputKey=key)
    # Initializing the bit vector with the file object
    bv = BitVector(filename=inputF)
    FILEOUT = open(outputF, 'wb')
    #Used for skipping the first 3 lines, and copying their values to the beginning of the output file
    newLine = 0
    newLine_bv = BitVector(intVal=10, size=8)
    while newLine != 3:
        bitVector = bv.read_bits_from_file(8)
        if bitVector == newLine_bv:
            newLine +=1
        bitVector.write_to_file(FILEOUT)
    while (bv.more_to_read):
        #Reading 64 bits from the file object and dividing it into two halves
        bitvec = bv.read_bits_from_file( 64 )
        [nextLE, nextRE] = bitvec.divide_into_two()
        if bitvec._getsize() > 0:
            #Checking to see if padding is necessary, and if so re-splitting the vector
            if bitvec._getsize() < 64:
                bitvec.pad_from_right(64 - bitvec._getsize())
                [nextLE, nextRE] = bitvec.divide_into_two()
            #iterating through all 16 rounds
            for round in range(16):
                LE = nextLE
                RE = nextRE
                newRE = RE.permute( expansion_permutation )
                out_xor = newRE.__xor__( round_keys[round] )
                afterSub = substitute(out_xor)
                readyToXorAgain = afterSub.permute(round_permutation)
                nextRE = readyToXorAgain.__xor__(LE)
                nextLE = RE
            #Combining the halfs and writing to the file as a hexstring
            output_bv = nextRE.__add__(nextLE)
            output_bv.write_to_file(FILEOUT)
    FILEOUT.close()
    return
```

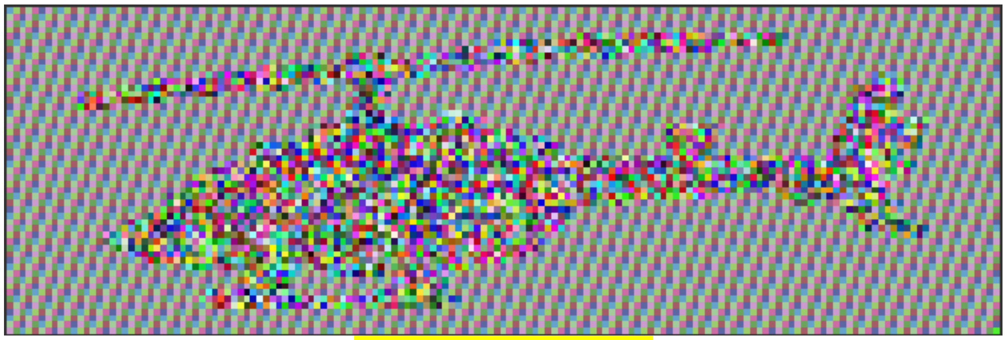
Terrence Randall

Test Code:



```
def test():
    print("encrypting image...")
    encrypt(inputF='image.ppm', keyF='testkey.txt', outputF='outImage.ppm')
```

Result: (output from encryption, and decryption)



<Image after encryption>

Code Summary: (explanation)

The encryption of the image is largely the same as the process for a text file. The only differences were:

1. The output file is opened as a binary file, to allow binary writing
2. The input file specially iterated 3 lines to copy the ppm header to the output without encrypting it
3. The bit-vector object (after encryption) is directly used to write the output file