ECE 404 Homework 2

Ranjan Behl (rbehl)

February 4, 2021

1. Problem 1: Encrypted Output

d04a94419ec6556c20029c83a277790c5c6380595291ecc23a40b90d60ae5b114dcefad2a37652e
80dbed6bea5ab59d92b8f043c65e1ced023bfe2aa6c4e162de19db8a75ff0f779baa3629395da7d
740e784fd3150db010f0054cd9f70a13ad6a553f954a79ca990dadc1a697ed8821548099cadedb2
d6572810b06df68150cd16af08948628fab087c8577826ee1e0ca728ea3def08044613e608e9ee
27cf91a7052f2d11e6a42b371c5216e296a5486887d331794502300e42cfe9b228da863420c7a9
d2eb3797bf08185451fc5948a61890e2fec008abe98af6a313ba886300a3041f4ca3f273f177fd
d95fb97cfd7724c196421848826c105892bfbbe47e64551e146fc6d7130d00a2dd01fa6b14a6fb
6fb054f843710ddd9a311d54882db94802ceac4fd454332747d76b4e6be9e614545db3e6a8517c
413628268c07aa64f7175ce8cd40a00a86fb279fed136146b2f863a0f54bb9407a21418641ba55
b6641e1acb69bdf816a2cf41479f80ddde5a4a43da9f53758f152f58bb5919b65d4a80250c259b
38f498f354223cbbcbe14e5408aadc581eb0d5b19ef8219fbe42edc4e9f0467826a5c1a8141af6
7f0a897b4f212e5ab49417b576aba488381be68fc72080ee3ed00b56152e2d7da477b92c98379b
694d4f466eb0d93d083fd62d36ef1ca7f3b4399af80559ffbe0bd48b6eb441a569d479f94a54cb
9ca816990971e229831db528e70972cae2f82df38026db9db5b118ff17df3a7621911b51626ab9
48dd95a777b4219b0ad0ab6180def71f24b42b23444d03b974681d583e07040d443d9365241e1f
a77e1b4684da92913a6ea9a2af407d586ddf8b242706e8775ced9fa520291bbafe441dfab3c4b5
d93cfe1654202d0b7ff5c381a6a2c489e2c756eb40b6b98482a49878d04f4422fcf43605826dd6
dc32cd8679e51bc800e3ae48673c19c5890c7eec8fc58775299ea756be20afff89395ac6b021f5
bb37c36e30f5948979c96b76537d8785721f1b9789e325d2e779c4e0859c093ba756c8998219cf
c497f0b7f66e259eebea3fba7a9ceed545ed833506d558c2dd9a8812ed9bdc69e9b0bdfbd51439
9d2a43be6bf50a2ddab68b3c3b449a430efdd46755871a8697737a7fd251de37390186a0c701ef
7839a2b2ee99a8d6aaf540aefd111c8507120fbc296ade7e0a30846b4ad461f2af4db654e8e000
8fa5a2fa42381d8350bd431d714c42f478ca43e3f31d4e2b77c4fea7b5fc92b55c18fd29ac06b7
8797333758a54e7aab3439dc079d168b7c416e23cd49084a57ff1c0974dd36102983521b30ecd7
fd1931201daab5059b8139f5a3017cd7fd1931201daab744fae27721dad95cd7fd1931201daab
abcce6cc5c4ddacecd7fd1931201daab462cde434ec9b646cd7fd1931201daab0013da3321192
b13b1bcd34158bc5878e3dbd126d0b7edf4cb345a27fa36e8df45ed30ec1b4cf954fd1fb2eb16
5e3fde33aab34a81ef30b95855c1917ea1826f0093dfae7a9e6124ac8036677dc75ddb8cc7954
8b5f6b673e2aaa2e74de559d17d4c3597eb793828ce2eba373130b87f4df0f0351aed3d07

2. Problem 1: Decrypted Output

Smartphone devices from the likes of Google, LG, OnePlus, Samsung and Xiaomi are in danger of compromise by cyber criminals after 400 vulnerable code sections were uncovered on Qualcomm's Snapdragon digital signal processor (DSP) chip, which runs on over 40

3. Explanation of the Code

The code I used is shown below:

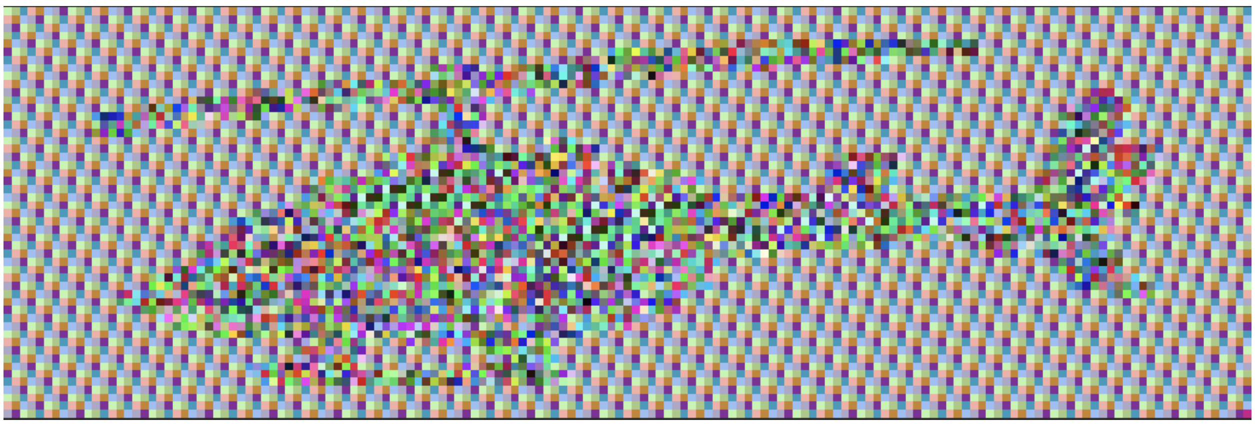
```
def encrypt(plainText,keyStr,cipherText):
    #break the plainText in 64 bit blocks
    bv = BitVector(filename=plainText)
    keyfp = open(keyStr,'r')
    cipherText = open(cipherText,'w')
    key = getEncryptionKey(keyfp)
    keyfp.close()
    roundKeys = extractRoundKeys( key )
    while (bv.more_to_read):
        bvRead = bv.read_bits_from_file( 64 )
        if(bvRead.length() != 64):
            #pad with zeros
            bvRead.pad_from_left(64 - bvRead.length())
        #break 64 bit vector into two 32 bit vectors and perform expansion permutation on r
        if bvRead.length() > 0: #bv.getsize() throws error for some reason
            #16 rounds
            bvHex = bvRead.get_bitvector_in_hex()
            #print("First 64 bit block before round 1:",bvHex)
            [leftHalf, rightHalf] = bvRead.divide_into_two()
            for i in range(16):
                originalRH = rightHalf # needed for the next left half
                #expand rightHalf to 48 bit
                newRH = rightHalf.permute(expansionPermutation)
                #XOR newRH with the roundkey to get new 48 bit
                outXor = newRH ^ roundKeys[i]
                #substitution with 8 s-boxes to get new 32 bit
                sBoxesOutput = substitute(outXor)
                #permutation with pBox
                rightHalf = sBoxesOutput.permute(pBoxPermutation)
                newRH = rightHalf ^ leftHalf
                newLH = originalRH # need og RH
                #swap to create the bitvector for the next round
                #reset bv to all zeros just to be safe
                bvRead.reset(0)
                leftHalf = newLH
                rightHalf = newRH
                #bvRead = newLH + newRH
                bvHex = bvRead.get_bitvector_in_hex()
                #print("First 64 bit block after round 1:",bvHex)
        #write the 64 encrypted block to the cipherText file
        bvRead = rightHalf + leftHalf
        print("64 Bit Block: ",bvRead.get_bitvector_in_hex())
        cipherText.write(bvRead.get_bitvector_in_hex())
    bv.close_file_object()
    cipherText.close()

def decrypt(cipherText,keyStr,plainText):
    #break the plainText in 64 bit blocks
    cipherText = open(cipherText.strip(),'r')
    data = cipherText.read()
    bv = BitVector(hexstring = data)
```

```
    cipherText.close()
    keyfp = open(keyStr,'r')
    plainText = open(plainText,'wb')
    key = getEncryptionKey(keyfp)
    roundKeys = extractRoundKeys( key )
    keyfp.close()
    j = 0
    k = 64
    #round key reverse
    roundKeys.reverse()
    while(k <= bv.length()):
        bvRead = bv[j:k]
        j = j + 64
        k = k + 64
        #print(bv_read)
        if(bvRead.length() != 64):
            #pad with zeros
            bvRead.pad_from_left(64 - bvRead.length())
        #break 64 bit vector into two 32 bit vectors and perform expansion permutation on r
        if bvRead.length() > 0:
            #16 rounds
            [leftHalf, rightHalf] = bvRead.divide_into_two()
            for i in range(16):
                originalRH = rightHalf # needed for the next left half
                #expand rightHalf to 48 bit
                newRH = rightHalf.permute(expansionPermutation)
                #XOR newRH with the roundkey to get new 48 bit
                outXor = newRH ^ roundKeys[i]
                #substitution with 8 s-boxes to get new 32 bit
                sBoxesOutput = substitute(outXor)
                #permutation with pBox
                rightHalf = sBoxesOutput.permute(pBoxPermutation)
                newRH = rightHalf ^ leftHalf
                newLH = originalRH # need og RH
                #swap to create the bitvector for the next round
                #reset bv to all zeros just to be safe
                bvRead.reset(0)
                leftHalf = newLH
                rightHalf = newRH
        #write the decrypted text to the decrypted file
        bvRead = rightHalf + leftHalf
        bvRead.write_to_file(plainText)
```

The code for the first problem depends on two functions(encrypt and decrypt) both of which are essentially the same function when it comes to the encryption logic due to the way a block cipher works. The encrypt functions takes three parameters,which are the file paths of the plainText,the key,and the soon to be created cipherText. To get the encryption key I resued the function that was provided in the example, similarly for the roundkey I resued the code provided. The plainText is converted to a bitvector and looped through in blocks of 64 bits(due to DES requirements). If the block that is read in doesnt have 64 bits the it is padded with zeros(left aligned to not increase the binary value) to make ita 64 bit block. After which the 64 bit block is divided into two halfs. Following which the round function is performed 16 times on the two halves. In the round function I reuse most of the provided code such as the expansion permutation, Sboxes the only new thing was creating the PBoxpermutation. At the end of a round the lefthalf and righthalf of the block is updated with newLH(which is the original righthalf) and the lefthalf is updated by newRH(which is the 32 bit xor result with orginal lefthalf and the output from the sboxes permutation). After the 16 rounds the right and left halves are swapped to create the correct encrypted block, which is then written to the cipherText.

The decryption is very simlar with the main difference being the roundkeys are used in reserve order and the way the cipherText is parsed is different than before. Since the cipherText is a hexstring I used array slicing to break the bitvector into 64 bit blocks. Everything else is the same as encrypt(). Also I wrote comments in the provided code to explain DES step by step.

4. Problem 2: Encrypted Image



5. Explanation of the Code

```
def encrypt(image,keyStr,image_enc):
    #break the image in 64 bit blocks
    bv = BitVector(filename=image)
    keyfp = open(keyStr,'r')
    image_enc = open(image_enc,'wb')
    #copy the header over to image_enc from imagefile
    idx = 0
    print("value of idx", idx)
    while(idx < 3):
        bvRead = bv.read_bits_from_file(8)
        #bvRead.get_bitvector_in_ascii()
        print(bvRead.get_bitvector_in_ascii())
        if(bvRead.get_bitvector_in_ascii() == '\n'):
            idx = idx + 1
        bvRead.write_to_file(image_enc)

    key = getEncryptionKey(keyfp)
    keyfp.close()
    print("value of idx", idx)
    roundKeys = extractRoundKeys( key )

    while (bv.more_to_read):
```

```
        bvRead = bv.read_bits_from_file( 64 )

        if(bvRead.length() != 64):
            #pad with zeros
            bvRead.pad_from_left(64 - bvRead.length())
        #break 64 bit vector into two 32 bit vectors and perform expansion permutation on r
        if bvRead.length() > 0: #bv.getsize() throws error for some reason
            #16 rounds
            #bvHex = bvRead.get_bitvector_in_hex()
            #print("First 64 bit block before round 1:",bvHex)
            [leftHalf, rightHalf] = bvRead.divide_into_two()
            for i in range(16):
                originalRH = rightHalf # needed for the next left half
                #expand rightHalf to 48 bit
                newRH = rightHalf.permute(expansionPermutation)
                #XOR newRH with the roundkey to get new 48 bit
                outXor = newRH ^ roundKeys[i]
                #substitution with 8 s-boxes to get new 32 bit
                sBoxesOutput = substitute(outXor)
                #permutation with pBox
                rightHalf = sBoxesOutput.permute(pBoxPermutation)
                newRH = rightHalf ^ leftHalf
                newLH = originalRH # need og RH
                #swap to create the bitvector for the next round
                #reset bv to all zeros just to be safe
                bvRead.reset(0)
                leftHalf = newLH
                rightHalf = newRH
                #bvRead = newLH + newRH
                #bvHex = bvRead.get_bitvector_in_hex()
                #print("First 64 bit block after round 1:",bvHex)
        #write the 64 encrypted block to the image_enc file
        #[leftHalf, rightHalf] = bvRead.divide_into_two()
        bvRead = rightHalf + leftHalf
        #print("64 Bit Block: ",bvRead.get_bitvector_in_hex())
        bvRead.write_to_file(image_enc)
    bv.close_file_object()
    image_enc.close()
```

The code used to encrypted the image is very similar to the code in problem 1. The main difference is that a ppm file has a header that doesnt need to be encrypted so I read the image file until I found three new line characters. At the same time I rewrote the header in the output file. Following which the remainder of the bitvector(the input image was converted to a bitvector) was read and the DES algorithm was performed on it.Every 64 bit block was encrypted and written to the output file after the header.