Code with Explanation for GGH Implementation on Images

'''

Authors: Yarala Hruthik Reddy, Kuruba Kiran Kumar, Surya Keesara

Title: GGH Cryptosystem Implementation on Image Files

'''

from PIL import Image

import random

import numpy as np

import base64

import binascii

import os

# This function generates a private and public key

def keyGen(dimension):

    privateKey = []

    print("Searching for a private key")

    ratio = 1

    '''

    while True:

        privateKey = np.random.randint(-10, 10, size=(dimension,dimension))

        ratio = hadamardRatio(privateKey, dimension)

        if(.9 <= ratio <= 1):

            print(privateKey)

            break

    '''

    privateKey = np.identity(dimension)

    print(privateKey)

    print("Searching for a public key")

    while True:

        uniMod = randUniMod(dimension)

        temp = np.matmul(uniMod, privateKey)

        ratio = hadamardRatio(temp, dimension)

        if ratio <= .1:

            publicKey = temp

            break

    print(publicKey)

    return privateKey, publicKey, uniMod

# This function returns the Hadamard Ratio of a matrix

def hadamardRatio(matrix, dimension):

    detOfLattice = np.linalg.det(matrix)

    detOfLattice = detOfLattice if detOfLattice > 0 else -detOfLattice

    mult = 1

    for v in matrix:

        mult = mult \* np.linalg.norm(v)

    hadRatio = (detOfLattice / mult) \*\* (1.0/dimension)

    return hadRatio

# This function returns a Random Unimodular matrix

def randUniMod(dimension):

    random\_matrix = [[np.random.randint(-10, 10,)

                      for \_ in range(dimension)] for \_ in range(dimension)]

    upperTri = np.triu(random\_matrix, 0)

    lowerTri = [[np.random.randint(-10, 10) if x <

                 y else 0 for x in range(dimension)] for y in range(dimension)]

    #Creating an upper trianglular and lower triangular matrices with diagonals as +1 or -1

    for r in range(len(upperTri)):

        for c in range(len(upperTri)):

            if(r == c):

                if bool(random.getrandbits(1)):

                    upperTri[r][c] = 1

                    lowerTri[r][c] = 1

                else:

                    upperTri[r][c] = -1

                    lowerTri[r][c] = -1

    uniModular = np.matmul(upperTri, lowerTri)

    return uniModular

# Converts images to black and white

def black\_and\_white(input\_image\_path, output\_image\_path):

   color\_image = Image.open(input\_image\_path)

   bw = color\_image.convert('L')

   bw.save(output\_image\_path)

# Loads images and copies the encoded image in a string

def loadImage(file\_path):

    with open(file\_path, "rb") as img\_file:

        my\_string = base64.b64encode(img\_file.read())

    print("Image Loaded")

    return my\_string

# Writes the encoded string to a file

def writeImageToString(filePath, string):

    newString = string.decode("utf-8")

    file = open(filePath, "w")

    file.write(str(newString))

    file.close()

# String to Image Converter

def writeStringToImage(filePath, imagePath):

    with open(filePath, "r") as f:

        string = f.read().replace("\n", "")

    imageData = base64.b64decode(string)

    with open(imagePath, "wb") as f:

        f.write(imageData)

# Writes text files

def writeTextBlocks(filePath, string):

    file = open(filePath, "w")

    file.write(str(string))

    file.close()

# Encoded text to Encrypted Ints

def encodedToBinaryToEncrypted(seekVal):

    encoded = []

    with open("ImageString\\imageString.txt", 'r') as f:

        encText = f.seek(seekVal)

        encText = f.read(10)

        for c in encText:

            encoded.append(base64.b64encode(bytes(c, "utf-8")))

    binaryMessage = []

    for i in range(len(encoded)):

        binaryMessage.append(binascii.a2b\_base64(encoded[i]))

    encryptedInts = []

    for i in range(len(encoded)):

        encryptedInts.append(int.from\_bytes(binaryMessage[i], byteorder='little'))

    return encryptedInts

# Encryption Function

def encrypt(encryptedInts, publicKey):

    cypherText = []

    for i in range(len(encryptedInts)):

        cypherText = np.matmul(encryptedInts, publicKey)

    return cypherText

# Decryption Function

def decrypt(cypherText, privateKey, uniModular):

    A = privateKey

    x = cypherText

    BPRIME = np.linalg.inv(A)

    BB = np.matmul(BPRIME, x)

    uniModularInv = np.linalg.inv(uniModular)

    m = np.round(np.matmul(BB, uniModularInv)).astype(int)

    return m

# Misc Methods

# Writes decrypted message to a file

def writeDecryptedMessage(filePath, message):

    file = open(filePath, "a+")

    for i in range(len(message)):

        letter = chr(abs(message[i]))

        file.write(letter)

    file.close()

'''

# Writes encrypted message to a file

def showEncryptedMessage(filePath, message):

    file = open(filePath, "a+")

    for i in range(len(message)):

        letter = chr(abs(int(message[i])))

        file.write(letter)

    file.close()

'''

# Writes Public key to a file

def writePublicKey(publicKey):

    file = open('PublicKey\\ggh\_block.txt', 'w')

    file.write("------BEGIN GGH PUBLIC KEY BLOCK -----\n")

    for row in range(len(publicKey)):

            for col in range(len(publicKey)):

                encoded = base64.b64encode(publicKey[row][col])

                file.write(str(encoded)[8:13])

            file.write("\n")

    file.write("\n--------END GGH PUBLIC KEY BLOCK -------")

    file.close()

# Writes the residue to the final message

def residueAdder(filePath, residueString):

    file = open(filePath, "a")

    for i in range(len(residueString)):

        letter = residueString[i]

        file.write(letter)

    file.close()

# Main Function

def main():

    dirs = ['BWImages', 'Decrypted', 'Encrypted', 'ImageString', 'PublicKey', 'Residue']

    for i in dirs:

        if not os.path.exists(i):

            os.mkdir(i)

            print("Directory ", i,  " Created ")

        else:

            print("Directory ", i,  " already exists")

    alice = keyGen(10)

    writePublicKey(alice[1])

    cypherTextFile = []

    inputFileName = 'InputImages\\' + input("Enter Input file name with relative path (InputImages\\fileName.jpg or .png): ")

    OutputMessageFile = 'Decrypted\\' + input("Enter the file name with relative path (Decrypted\\fileName.txt) to output the decrypted message to: ")

    black\_and\_white(inputFileName, 'BWImages\\bwImage.jpg')

    stringEncoded = loadImage("BWImages\\bwImage.jpg")

    writeImageToString("ImageString\\imageString.txt", stringEncoded)

    print(len(stringEncoded))

    print("\n-------------------------Encrypting------------------------\n")

    seekVal = 0

    while True:

        encryptedInts = encodedToBinaryToEncrypted(seekVal)

        if (seekVal > len(stringEncoded) - 10):

            residueString = stringEncoded[-(len(stringEncoded) - seekVal):]

            break

        else:

            seekVal = seekVal + 10

        bob = encrypt(encryptedInts, alice[1])

        cypherTextFile.append(bob)

    residueString = residueString.decode("utf-8")

    writeTextBlocks("Encrypted\\cypherTextFile.txt", cypherTextFile)

    writeTextBlocks("Residue\\residueText.txt", residueString)

    '''

    print("\n----------------Encrypted Message Print-------------------\n")

    seekVal = 0

    while True:

        if (seekVal >= len(cypherTextFile)):

            break

        else:

            showEncryptedMessage("Encrypted\\encryptedMessage.txt", cypherTextFile[seekVal])

        seekVal = seekVal + 1

    residueAdder("Encrypted\\encryptedMessage.txt", residueString)

    writeStringToImage("Encrypted\\encryptedMessage.txt", "Encrypted\\encryptedImage.jpg")

    '''

    print("\n------------------------Decrypting---------------------------\n")

    seekVal = 0

    decryptedTextFile = []

    while True:

        if (seekVal >= len(cypherTextFile)):

            break

        else:

            aliceReceives = decrypt(cypherTextFile[seekVal], alice[0], alice[2])

            decryptedTextFile.append(aliceReceives)

        seekVal = seekVal + 1

    writeTextBlocks("Decrypted\\decryptedTextFile.txt", decryptedTextFile)

    print("\n---------------Decrypted Message Printing-------------------\n")

    seekVal = 0

    while True:

        if (seekVal >= len(decryptedTextFile)):

            break

        else:

            writeDecryptedMessage(OutputMessageFile, decryptedTextFile[seekVal])

        seekVal = seekVal + 1

    residueAdder("Decrypted\\decryptedMessage.txt", residueString)

    writeStringToImage(OutputMessageFile, "Decrypted\\decryptedImage.jpg")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Libraries:-**

Image library to open and load images. Random library to generate random numbers. NumPy library for linear algebra calculations such as matrix multiplications. Base64 library for encoding and decoding strings. Binascii library to convert characters to integers. OS library for directory and file path handling.

**Key Generation:-**

There was an attempt to generate a nearly orthogonal basis as a private key (Commented Code in the **keyGen()** function) but the success rate of decrypting back the original message was around 40% due to floating point computation errors. Instead, we chose a perfectly orthogonal basis i.e., Identity matrix of respective dimension as the private key. The public key was generated by multiplying a random unimodular matrix with the private key. The public key is accepted only if its Hadamard Ratio is close to zero or if the public key is a nearly parallel basis. The function returns the private key, the public key and the unimodular matrix used.

**Image Handling:-**

Any input images were converted to grayscale to reduce the size of the image, **black\_and\_white()** function does this operation. The grayscale images were then read and encoded into a string, **loadImage()** function does this operation.

**Encryption:-**

The encoded text was converted to encrypted integers by through ASCII conversions, **encodedToBinaryToEncrypted()** function achieves this operation. The encrypted integers were then sent to the **encrypt()** function to generate the final cypher text.

**Decryption:-**

The cypher text along with the private key and the unimodular matrix were sent to the **decrypt()** function. The inverses of the private key and the unimodular matrix were generated and the inverse of the private key was multiplied with the cypher text. This was then multiplied with the inverse of unimodular matrix to get back the original matrix.

**Helper Functions:-**

**hadamardRatio()** and **randUniMod()** functions are the helper functions for the **keyGen()** function. **writeImageToString()** function writes the image string to a file and the **writeStringToImage()** function writes the string back to the image. **writeTextBlocks()** functions writes the encrypted and decrypted integer messages to text files. **writeDecryptedMessage()** function converts the decrypted integers into a string and writes them in a text file. **writePublicKey()** function writes the public key to a text file. **residueAdder()** function adds the residue characters back to the original message.

**NOTE:-**

* Image input should be of size of 100 width and 100 height. Only grayscale encryption and decryption are done. Could be extended for colour images too but will be computationally heavy.
* Image string was broken down into individual strings of 10 characters each as the key dimensions were also 10 in this case. Higher keys and higher individual length of strings can be used but it is computationally heavy. We found 10 sized strings and 10 sized keys to be perfect for our computations.
* The residueAdder() is an optional function to take care of the residue bits, but the loss of those bits will not impact the image significantly.
* We tried to visually show the encrypted message as an image file but the mapping of the encrypted message integers to the respective ASCII characters is not possible in most of the cases and results in errors rendering the encrypted image.