Secure Image Processing using Paillier Homomorphic Encryption

Abstart:

Securing data online against attackers is a big task to acheive. But we can acheive this with cryptographic algorithms. Cryptographic algorithms like RSA, AES etc provide best data protection. These are easier to compute and diffucult to crack. But there is a drawback in these encryption algorithms i.e they lack secure processing ability. If we need to process conventionally encrypted data we need a stack of operations needed to be done every time like

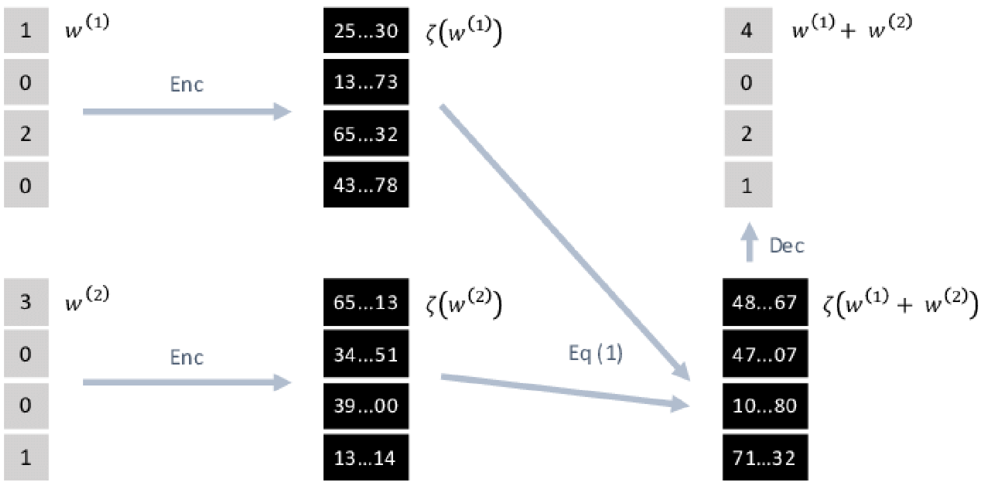
1. Decrypt the encrypted data
2. Process it and make changes needed on it
3. Encrypt the resultant data

This process is time consuming. Even if we opt to tolerate it, there it needs a huge data transfer everytime if the data is located on cloud. So this solution is both time consuming as well as overburden. So we need an encryption mechanism that can process the encrypted data without decrypting it. And the solution here is Homomorphic Encryption. This can process the encrypted data without need for decrypting it. These resulting computations are left in an encrypted form which, when decrypted, result in an identical output to that produced had the operations been performed on the unencrypted data.

This project uses the Paillier Homomorphic(Partial) System to encrypt the image data and perform image processing techniques(only few) on it.

Introduction & Literature:

1. **Homomorphic Encryption** : It is a form of encryption that permits users to perform computations on it's encrypted data without decrypting it. These resulting computations are left in an encrypted form which, when decrypted, result is identical output to that produced had the operations been performed on unencrypted data.



2. **Paillier Encryption :** The Paillier cryptosystem is a probabilistic asymmetric algorithm for public key cryptography. The problem of computing n-th residue classes is believed to be computationally difficult. The decisional composite residuosity assumptions in the intractability hypothesis upon which this Paillier crytosystem is based. This cryptosystem is partially homomorphic that allows addition operations on it.

$$E(m1)+E(m2) \rightarrow E(m1+m2)$$

Algorithm:

1. Key Generation:

- Choose two large numbers p and q randomly and independently of each other such that $gcd(pq, (p-1)(q-1))=1$.

- Compute $n=pq$ and $\lambda = lcm(p-1, q-1)$. lcm is least comman multiple.

- Select random integer g where $g \in Z_{n^2}^*$
- Ensure n divides the order of g by checking the existance of the following modular multiplicative inverse : $\mu = (L(g^\lambda mod\, n^2))^{-1} mod\, n$ where function L is defined as $L(x)=(x-1)/n$
- Public key is (n, g)

- Private Key is (λ, μ)

2. Encryption:

- Let m be the message to be encrypted where $0 \le m < n$

- Select random r where $0 < r < n$ and $r \in Z_n^*$ (i.e ensure $gcd(r, n)=1$)

- Compute cipher text as $c = g^m . r^n mod\, n^2$

3. Decryption:

- Let c be the ciphertext to decrypt, where $c \in Z_{n^2}^*$

- Compute the plaintext as $m = L(c^\lambda mod\, n^2). \mu mod\, n$

3. **Homomorphic Properties:**

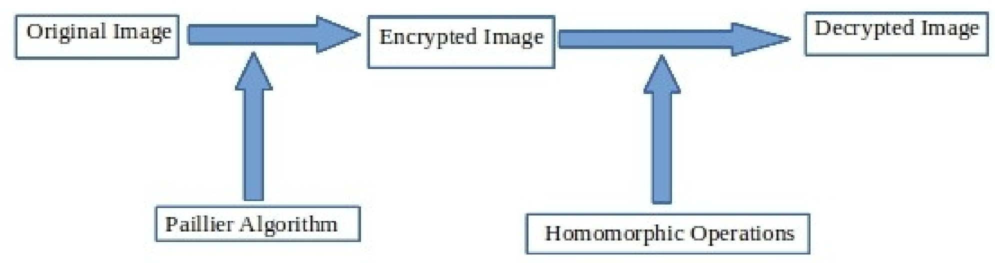
1. Addition of Cipher Data:

$$D(E(m_1, r_1). E(m_2, r_2) mod\, n^2) = m_1 + m_2 mod\, n$$

2. Multiplication of Plain text on Cipher text:

$$D(E(m_{1,} r1)^{m_2} mod\, n^2) = m_1 . m_2 mod\, n$$

4. **Project Details**
   1. Encrypting images data at pixel level and storing cipher value at each position.
   2. Both RGB, Gray scale images are involved.
   3. Allowing image processing techniques like increasing brightness, color etc.
   4. Again decrypting to produce output image.



Software Requirements Specification:

1. **Purpose:**Due to increase in demand for data security and secured processing needs, there is a huge demand for Homomorphic Encryption. This project serves as a template for implementing homomorphic encryption for secured image processing.

2. **Definitions:**
   i. **Homomorphic Encryption:** Post quantum data encryption technique that provides homomorphic operations to be done.

   ii. **Paillier Encryption:** It is a probabilistic asymmentric algorithm for public key homomorphic encryption.

   iii. **Modular Multiplicative Inverse:** A modular multiplicative inverse of an integer a is an integer x such that the product ax is congruent to 1 with respect to modulus m.

   $$ax \equiv 1 (mod\, m)$$

   iv. **Image :** It is an n dimensional array with values ranging between 0 and 255.

3. **Assumptions:**
   i. Assume that the problem of computing n-th residue classes is believed to be computationally difficult.

ii. Assume the random number generation used in computers is really random not predictable.

4. **Scope:**

i. The scope of the project is restricted to python language only, but the similar implementations can be made on other languages too.

ii. The project only deals with images that are GRAY (2D), RGB(3D) and RGBA(4D).

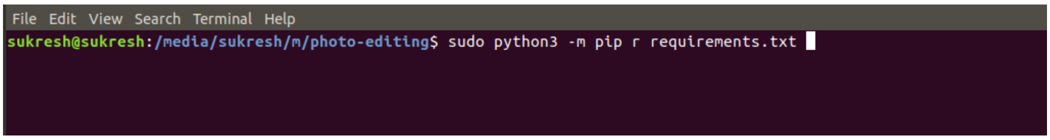
5. **Product Functions:**

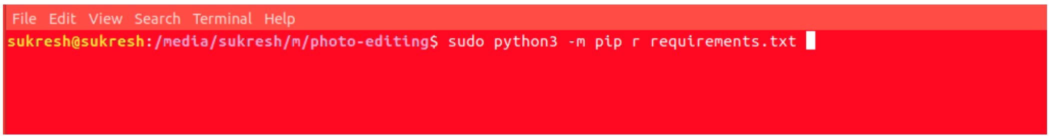
i. **Encrypt/Decrypt Image:** By using paillier encryption, encrypt/decrypt the imageand maintain a copy in memory for usage. Takes parameters as $(pk, image)$ for encryption and $(priv, pk, image)$ for decryption.

```
sukresh@sukresh:~/Documents/E3-project/code/paillier/paillier$ python3 main.py
[[ 51  49  45 255]
 [ 51  49  45 255]
 [ 55  55  50 255]
 ...
 [ 79  77  70 255]
 [ 79  77  70 255]
 [ 79  77  70 255]]
[[239419431306  60972494969 176378814376  21299446521]
 [239419431306  60972494969 176378814376  21299446521]
 [124013111899 124013111899 386346059178  21299446521]
 ...
 [376175579619 197728643282 281614654224  21299446521]
 [376175579619 197728643282 281614654224  21299446521]
 [376175579619 197728643282 281614654224  21299446521]]
sukresh@sukresh:~/Documents/E3-project/code/paillier/paillier$
```

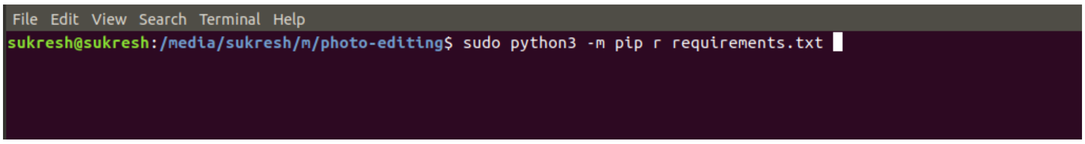
ii. **Brightness Increase:** Increase image brightness by pixel level on the whole image. Takes parameter as $(pk, image, filter\_strength)$

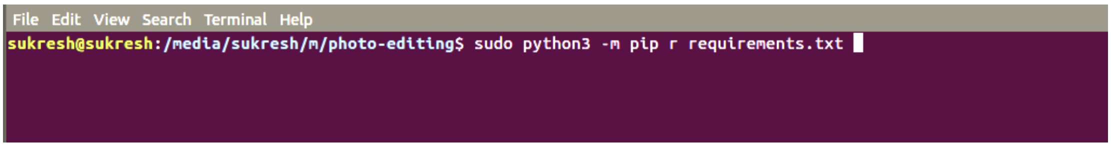
iii. **Increase Color:** Using the same approach as brightness but only on the specified color channel increase the values.. This is only applicable to RGB or RGBA images. Takes parameters as $(pk, image, \text{color\_channel}, filter\_strength)$



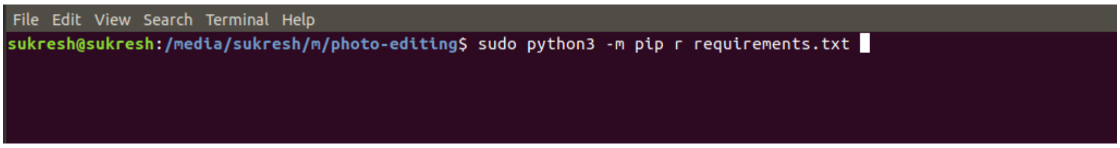


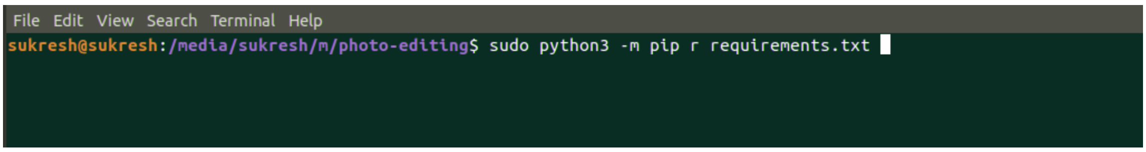
iv. Multiply By Constant: Using multiplication as a homomorphic operation multiply all the pixel values by specified value. Takes parameters as $(pk, image, \text{filter\_strength})$



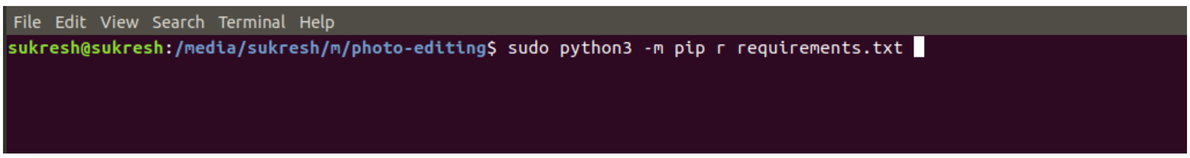


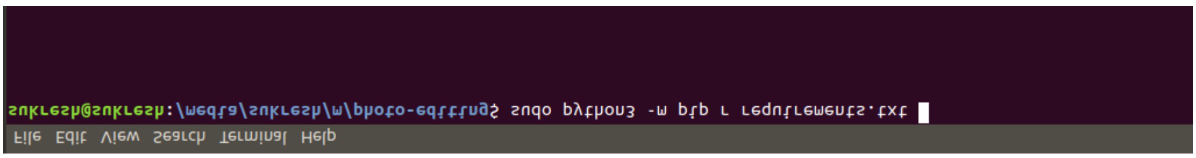
v. Swap Colors : Just by using array manipulation techniques change the values in one color channel to another color channel and vice versa. This is only applicable to RGB or RGBA images. Takes parameters as $(pk, image, \text{color\_channel1}, \text{color\_channel2})$



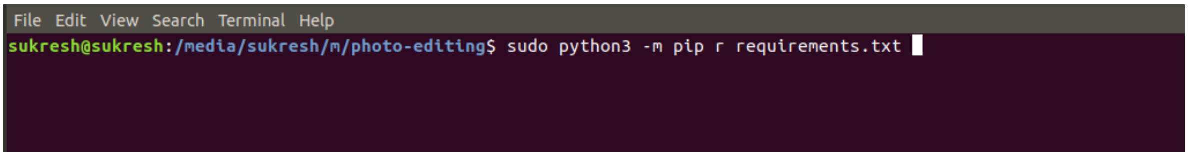


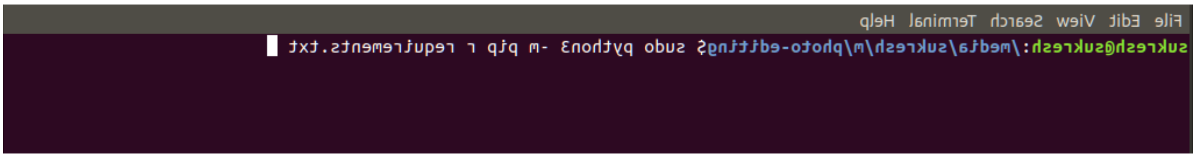
vi. Flip Image: By using simple array manipulation techniques, flip the image top to bottom and vice versa. Takes parameters as $(pk, image)$





vii. **Mirror Image:** By using simple array manipulation techniques mirror the image by interchanging rightside and leftside pixel values.Takes parameters as $(pk, image)$





6. **External Libraries:**
   i. This project uses multiple libraries for hardware acceleration and clean code writing. They are
      i. **python3** : A dynamic types interpreted language.
      ii. **numpy** : A python n-dimensional array manipulation program with hardware acceleration.
      iii. PIL : A python library for importing images and converting them into n-dimensional arrays.
   ii. This project develops it's own library for prime number generation, inverse mod calculation and exponent modulus operation.

7. **Performance Requirements:**
   i. This project uses atleast 4GB of RAM and 8GB of HardDisk for better performance.
   ii. It requires 64-bit architecture system.

8. **Limitations:**
   i. This project is designed for cpu performance utilization but cpu alone can't handle large images and large inverse mod calculations.
   ii. This paillier encryption allows only addition and multiplication as homomorphic encryption, but it can't afford subtraction and division. So most image processing techniques are not possible to create.

Proposed Work:

1. **Prime Number Generation:** Will generate an integer of b bits that is probably prime (after k trials). Reasonably fast on current hardware for values of up to around 512 bits. Generate a random number in range 2**(bits-1) and 2**bits. Then check if number is probably prime using Robin Miller witness test. If it is still prime after k trails then it is considered as real prime.

2. **Generate Keypair :** First generate two prime number p,q and claculate $n=p*q$. Then our public key pair is (n) . For private key find $l=(p-1)*(q-1)$ and then find inverse modulus of l and n, the result is m. So our private key is (l, m)

3. **Encrypt:** Using public key one can easily find cipher text as $cipher=(pow(r, plain, n^2)*x)\%n^2$. Where r is another random prime number and $x=pow(r, n, n^2)$
4. **Decrypt**: Using both public key and private key we can decrypt the cipher as $plain=((x // pub.n)*priv.m)\%n$ where $x=pow(cipher, priv.l, pub.n^2)-1$
5. **Add** : We can add two cipher texts as another cipher text like $D(E(m_1, r_1).E(m_2, r_2) mod\, n^2)=m_1+m_2 mod\, n$
6. **Multiplication:** We can multiply a plain text to a cipher text as $D(E(m_{1,} r1)^{m_2} mod\, n^2)=m_1.m_2 mod\, n$
7. **Encrypt Image:** Consider an image as a n-dimensional array like GRAY(2D), RGB(3D) and RGBA(4D). At every block level i.e pixel level encrypt the image and make it dimensions priserved.
8. **Brightness:** Increasing brightness is an addition of some constant k on every pixel. So by using paillier homomorphic addition, add some constant k on every pixel of encrypted image.
9. **Increase Color:** Increasing color is a similar process like increasing brightness, rather than applying on every pixel and color channel we apply the addition on only a specified color channel.
10. **Mirror Image:** It is a simple array manipulation process, where left ward pixels are interchanged with the right ward pixels.
11. **Flip Image:** This a simple array manipulation process, where topward pixels are interchanged with the bottom pixels, so as to create a flipped image.
12. **Swap Colors:** It is a simple array manipulation where rather than pixel values color channels are interchanged.
13. **Multiply by Const:** It is a multiplication of a constant k on every pixel using paillier homomorphic multiplication process.

Implementation:

primes.py

```
import random
import sys
def ipow(a, b, n):
	"""calculates (a**b) % n via binary exponentiation, yielding itermediate
results as Rabin-Miller requires"""
	A = a = int(a % n)
	yield A
	t = 1
	while t <= b:
	t <<= 1

	# t = 2**k, and t > b
	t >>= 2

	while t:
	A = (A * A) % n
	if t & b:
		A = (A * a) % n
	yield A
	t >>= 1
```

```
def rabin_miller_witness(test, possible):
      """Using Rabin-Miller witness test, will return True if possible is
       definitely not prime (composite), False if it may be prime."""
      return 1 not in ipow(test, possible-1, possible)

smallprimes = (2,3,5,7,11,13,17,19,23,29,31,37,41,43,
               47,53,59,61,67,71,73,79,83,89,97)

def default_k(bits):
      return max(40, 2 * bits)

def is_probably_prime(possible, k=None):
      if possible == 1:
      return True
      if k is None:
      k = default_k(possible.bit_length())
      for i in smallprimes:
      if possible == i:
            return True
      if possible % i == 0:
            return False
      for i in range(int(k)):
      test = random.randrange(2, possible - 1) | 1
      if rabin_miller_witness(test, possible):
            return False
      return True

def generate_prime(bits, k=None):
      """Will generate an integer of b bits that is probably prime
       (after k trials). Reasonably fast on current hardware for
       values of up to around 512 bits."""
      assert bits >= 8

      if k is None:
      k = default_k(bits)

      while True:
      possible = random.randrange(2 ** (bits-1) + 1, 2 ** bits) | 1
      if is_probably_prime(possible, k):
            return possible
```

paillier.py

```
import math
import primes
import numpy as np
from PIL import Image

class PrivateKey(object):

      def __init__(self, p, q, n):
      self.l = (p-1) * (q-1)
      self.m = paillier.invmod(self.l, n)

      def __repr__(self):
      return '<PrivateKey: {} {}>'.format(self.l, self.m)
```

```
class PublicKey(object):

      @classmethod
      def from_n(cls, n):
            return cls(n)

      def __init__(self, n):
            self.n = n
            self.n_sq = n * n
            self.g = n + 1

      def __repr__(self):
            return '<PublicKey: {}>'.format(self.n)

class paillier:
      def __init__(self):
            self.prime_r = 1

      def invmod(a, p, maxiter=1000000):

            if a == 0:
                  raise ValueError('0 has no inverse mod {}'.format(p))
            r = a
            d = 1
            for i in range(min(p, maxiter)):
                  d = ((p // r + 1) * d) % p
                  r = (d * a) % p
                  if r == 1:
                        break
            else:
                  raise ValueError('{} has no inverse mod {}'.format(a, p))
            return d

      def modpow(self, base, exponent, modulus):

            result = 1
            while exponent > 0:
                  if exponent & 1 == 1:
                        result = (result * base) % modulus
                  exponent = exponent >> 1
                  base = (base * base) % modulus
            return result

      def generate_keypair(self, bits):
            p = primes.generate_prime(bits / 2)
            q = primes.generate_prime(bits / 2)
            n = p * q
            return PrivateKey(p, q, n), PublicKey(n)

      def encrypt(self, pub, plain):
            if(self.prime_r == 1):
                  while True:
                        self.prime_r =
                        primes.generate_prime(int(round(math.log(pub.n, 2))))
                        if self.prime_r > 0 and self.prime_r < pub.n:
                              break
            x = pow(self.prime_r, pub.n, pub.n_sq)
            cipher = (pow(pub.g, plain, pub.n_sq) * x) % pub.n_sq
            return cipher
```

```
    def e_add(self, pub, a, b):
        """Add one encrypted integer to another"""
        return a * b % pub.n_sq

    def e_add_const(self, pub, a, n):
        """Add constant n to an encrypted integer"""
        return a * self.modpow(pub.g, n, pub.n_sq) % pub.n_sq

    def e_mul_const(self, pub, a, n):
        """Multiplies an encrypted integer by a constant"""
        a = int(a)
        return self.modpow(a, n, pub.n_sq)

    def decrypt(self, priv, pub, cipher):
        cipher = int(cipher)
        x = pow(cipher, priv.l, pub.n_sq) - 1
        plain = ((x // pub.n) * priv.m) % pub.n
        return plain

    def open_image(self,path):
        image = np.array(Image.open(path))
        return image

    def save_image(self,image):
        image = image.astype(np.uint8)
        im = Image.fromarray(image)
        if(len(image.shape) == 3):
            if(image.shape[2] == 4):
                im.save("output.png")
            else:
                im.save("output.jpg")
        else:
            im.save("output.png")

    def encrypt_image_temp(self, pub, image):
        img = []
        dimens = image.shape
        for row in image.flatten():
            img.append(self.encrypt(pub,int(row)))
        if(len(dimens) == 3):
            return np.array(img).reshape(dimens[0], dimens[1], -1)
        elif(len(dimens) == 2):
            return np.array(img).reshape(dimens[0], -1)
        else:
            return None
    def encrypt_image(self, pub, image):
        string = "Pillier homomorphic encryption is very difficult to
generate so it uses a lot resources so of no of channels*width*height >=
2488320 we are stoping it early so that there won't be any waiting......."
        if(np.prod(image.shape) >= 2488320):
            print(string)
            return
        return self.encrypt_image_temp(pub, image)
```

```
    def decrypt_image(self, priv, pub, image):
        img = []
        dims = image.shape
        for pixel in image.flatten():
            temp = self.decrypt(priv, pub, pixel)
            if(temp > 255):
                img.append(255)
            else:
                img.append(temp)
        if(len(dims) == 4):
            return np.array(img).reshape(dims[0], dims[1], dims[2], -1)
        elif(len(dims) == 3):
            return np.array(img).reshape(dims[0], dims[1], -1)
        elif(len(dims) == 2):
            return np.array(img).reshape(dims[0], -1)
        else:
            return None

    def brightness(self, pub, image, value):
        img = []
        dims = image.shape
        for pixel in image.flatten():
            img.append(self.e_add_const(pub, pixel, value))
        if(len(dims) == 3):
            return np.array(img).reshape(dims[0], dims[1], -1)
        elif(len(dims) == 2):
            return np.array(img).reshape(dims[0], -1)
        else:
            None

    def increase_color(self, pub, image, color, value):
        color_schema = {
            "red" : 0,
            "green" : 1,
            "blue" : 2,
            "luminance" : 3
        }
        img = []
        if len(image.shape) == 2:
            return "You con't change "+color+" channel of a gray scale
image"
        elif(len(image.shape) >= 3):
            color_channel = image[:,:,color_schema[color]]
            copy = color_channel.copy()
            for i in copy.flatten():
                img.append(self.e_add_const(pub, i, value))
            color_channel = np.array(img).reshape(color_channel.shape[0],
-1)

            image[:,:,color_schema[color]] = color_channel
            return image

    def mirroring_image(self, pub, image):
        i = 0
        j = image.shape[1]-1
        while(i < j and i+j < image.shape[1]):
            temp = image[:,i].copy()
            image[:,i] = image[:,j].copy()
            image[:,j] = temp.copy()
            i+=1
            j-=1
        return image
```

```
    def flip_image(self, pub, image):
        i = 0
        j = image.shape[0]-1
        while(i < j and i+j < image.shape[0]):
            temp = image[i].copy()
            image[i] = image[j].copy()
            image[j] = temp.copy()
            i+=1
            j-=1
        return image

    def swap_colors(self, pub, image, color1, color2):
        color_schema = {
            "red" : 0,
            "green" : 1,
            "blue" : 2,
            "luminance" : 3
        }
        img = []
        if len(image.shape) == 2:
            return "You con't change "+color+" channel of a gray scale 
image"
        else:
            first_channel = image[:,:,color_schema[color1]].copy()
            second_chennel = image[:,:,color_schema[color2]].copy()
            image[:,:,color_schema[color2]] = first_channel
            image[:,:,color_schema[color1]] = second_chennel
            return image
    def multiply_by_const(self, pub, image, e):
        img = []
        copy = image.copy()
        dims = image.shape
        for pixel in copy.flatten():
            img.append(self.e_mul_const(pub, pixel, e))
        if(len(dims) == 2):
            return np.array(img).reshape(dims[0], -1)
        elif(len(dims) == 3):
            return np.array(img).reshape(dims[0], dims[1], -1)
        else:
            return np.array(img).reshape(dims[0], dims[1], dims[2] -1)
```

Result:

This project is a simple usecase of homomorphic encryption only limited to image processing. This provides facility for secured processing of image data in the encrypted format.

Improvements:

- **i.** Update the project to support parallel processing needs using dask(python library for parallel processing)
- **ii.** Update it to support all types of images and sizes

References:

1. https://en.wikipedia.org/wiki/Paillier_cryptosystem
2. https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
3. https://github.com/mikeivanov/paillier

4. https://numpy.org/doc/