|  |  |
| --- | --- |
|  | #Nethra’s file  #Work on progress. |
|  | import docopt |
|  | import numpy as np  import cv2 |
|  |  |
|  |  |
|  | class SteganographyException(Exception): |
|  | pass |
|  |  |
|  |  |
|  | class LSBSteg(): |
|  | def \_\_init\_\_(self, im): |
|  | self.image = im |
|  | self.height, self.width, self.nbchannels = im.shape |
|  | self.size = self.width \* self.height |
|  |  |
|  | self.maskONEValues = [1,2,4,8,16,32,64,128] |
|  | #Mask used to put one ex:1->00000001, 2->00000010 .. associated with OR bitwise |
|  | self.maskONE = self.maskONEValues.pop(0) #Will be used to do bitwise operations |
|  |  |
|  | self.maskZEROValues = [254,253,251,247,239,223,191,127] |
|  | #Mak used to put zero ex:254->11111110, 253->11111101 .. associated with AND bitwise |
|  | self.maskZERO = self.maskZEROValues.pop(0) |
|  |  |
|  | self.curwidth = 0 # Current width position |
|  | self.curheight = 0 # Current height position |
|  | self.curchan = 0 # Current channel position |
|  |  |
|  | def put\_binary\_value(self, bits): #Put the bits in the image |
|  | for c in bits: |
|  | val = list(self.image[self.curheight,self.curwidth]) #Get the pixel value as a list |
|  | if int(c) == 1: |
|  | val[self.curchan] = int(val[self.curchan]) | self.maskONE #OR with maskONE |
|  | else: |
|  | val[self.curchan] = int(val[self.curchan]) & self.maskZERO #AND with maskZERO |
|  |  |
|  | self.image[self.curheight,self.curwidth] = tuple(val) |
|  | self.next\_slot() #Move "cursor" to the next space |
|  |  |
|  | def next\_slot(self):#Move to the next slot were information can be taken or put |
|  | if self.curchan == self.nbchannels-1: #Next Space is the following channel |
|  | self.curchan = 0 |
|  | if self.curwidth == self.width-1: #Or the first channel of the next pixel of the same line |
|  | self.curwidth = 0 |
|  | if self.curheight == self.height-1:#Or the first channel of the first pixel of the next line |
|  | self.curheight = 0 |
|  | if self.maskONE == 128: #Mask 1000000, so the last mask |
|  | raise SteganographyException("No available slot remaining (image filled)") |
|  | else: #Or instead of using the first bit start using the second and so on.. |
|  | self.maskONE = self.maskONEValues.pop(0) |
|  | self.maskZERO = self.maskZEROValues.pop(0) |
|  | else: |
|  | self.curheight +=1 |
|  | else: |
|  | self.curwidth +=1 |
|  | else: |
|  | self.curchan +=1 |
|  |  |
|  | def read\_bit(self): #Read a single bit int the image |
|  | val = self.image[self.curheight,self.curwidth][self.curchan] |
|  | val = int(val) & self.maskONE |
|  | self.next\_slot() |
|  | if val > 0: |
|  | return "1" |
|  | else: |
|  | return "0" |
|  |  |
|  | def read\_byte(self): |
|  | return self.read\_bits(8) |
|  |  |
|  | def read\_bits(self, nb): #Read the given number of bits |
|  | bits = "" |
|  | for i in range(nb): |
|  | bits += self.read\_bit() |
|  | return bits |
|  |  |
|  | def byteValue(self, val): |
|  | return self.binary\_value(val, 8) |
|  |  |
|  | def binary\_value(self, val, bitsize): #Return the binary value of an int as a byte |
|  | binval = bin(val)[2:] |
|  | if len(binval) > bitsize: |
|  | raise SteganographyException("binary value larger than the expected size") |
|  | while len(binval) < bitsize: |
|  | binval = "0"+binval |
|  | return binval |
|  |  |
|  | def encode\_text(self, txt): |
|  | l = len(txt) |
|  | binl = self.binary\_value(l, 16) #Length coded on 2 bytes so the text size can be up to 65536 bytes long |
|  | self.put\_binary\_value(binl) #Put text length coded on 4 bytes |
|  | for char in txt: #And put all the chars |
|  | c = ord(char) |
|  | self.put\_binary\_value(self.byteValue(c)) |
|  | return self.image |
|  |  |
|  | def decode\_text(self): |
|  | ls = self.read\_bits(16) #Read the text size in bytes |
|  | l = int(ls,2) |
|  | i = 0 |
|  | unhideTxt = "" |
|  | while i < l: #Read all bytes of the text |
|  | tmp = self.read\_byte() #So one byte |
|  | i += 1 |
|  | unhideTxt += chr(int(tmp,2)) #Every chars concatenated to str |
|  | return unhideTxt |
|  |  |
|  | def encode\_image(self, imtohide): |
|  | w = imtohide.width |
|  | h = imtohide.height |
|  | if self.width\*self.height\*self.nbchannels < w\*h\*imtohide.channels: |
|  | raise SteganographyException("Carrier image not big enough to hold all the datas to steganography") |
|  | binw = self.binary\_value(w, 16) #Width coded on to byte so width up to 65536 |
|  | binh = self.binary\_value(h, 16) |
|  | self.put\_binary\_value(binw) #Put width |
|  | self.put\_binary\_value(binh) #Put height |
|  | for h in range(imtohide.height): #Iterate the hole image to put every pixel values |
|  | for w in range(imtohide.width): |
|  | for chan in range(imtohide.channels): |
|  | val = imtohide[h,w][chan] |
|  | self.put\_binary\_value(self.byteValue(int(val))) |
|  | return self.image |
|  |  |
|  |  |
|  | def decode\_image(self): |
|  | width = int(self.read\_bits(16),2) #Read 16bits and convert it in int |
|  | height = int(self.read\_bits(16),2) |
|  | unhideimg = np.zeros((width,height, 3), np.uint8) #Create an image in which we will put all the pixels read |
|  | for h in range(height): |
|  | for w in range(width): |
|  | for chan in range(unhideimg.channels): |
|  | val = list(unhideimg[h,w]) |
|  | val[chan] = int(self.read\_byte(),2) #Read the value |
|  | unhideimg[h,w] = tuple(val) |
|  | return unhideimg |
|  |  |
|  | def encode\_binary(self, data): |
|  | l = len(data) |
|  | if self.width\*self.height\*self.nbchannels < l+64: |
|  | raise SteganographyException("Carrier image not big enough to hold all the datas to steganography") |
|  | self.put\_binary\_value(self.binary\_value(l, 64)) |
|  | for byte in data: |
|  | byte = byte if isinstance(byte, int) else ord(byte) # Compat py2/py3 |
|  | self.put\_binary\_value(self.byteValue(byte)) |
|  | return self.image |
|  |  |
|  | def decode\_binary(self): |
|  | l = int(self.read\_bits(64), 2) |
|  | output = b"" |
|  | for i in range(l): |
|  | output += chr(int(self.read\_byte(),2)).encode("utf-8") |
|  | return output |
|  |  |
|  |  |
|  | def main(): |
|  | args = docopt.docopt(\_\_doc\_\_, version="0.2") |
|  | in\_f = args["--in"] |
|  | out\_f = args["--out"] |
|  | in\_img = cv2.imread(in\_f) |
|  | steg = LSBSteg(in\_img) |
|  |  |
|  | if args['encode']: |
|  | data = open(args["--file"], "rb").read() |
|  | res = steg.encode\_binary(data) |
|  | cv2.imwrite(out\_f, res) |
|  |  |
|  | elif args["decode"]: |
|  | raw = steg.decode\_binary() |
|  | with open(out\_f, "wb") as f: |
|  | f.write(raw) |
|  |  |
|  |  |
|  | if \_\_name\_\_=="\_\_main\_\_": |
|  | main() |