National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS 490: Advanced Topics in Computing (Computer Vision)

Class: BSCS 9C

Amal Saqib (282496)

Hamza Mushtaque (288114)

Assignment # 1 : Steganography Image Hiding Using Python and OpenCV

Date: 8th March 2023

1. Explain the importance of steganography techniques in data security and discuss three real-life scenarios where steganography can be used.

Importance

Steganography is a technique that allows users to hide secret data within a seemingly harmless carrier file, such as an image, video, or audio file. The importance of steganography techniques in data security lies in their ability to provide an additional layer of security to sensitive information by making it difficult for unauthorized individuals to detect or access the hidden data. Individuals can use steganography to hide secret messages within images or videos that can be transmitted over the internet without arousing suspicion. Steganography can also help protect sensitive data from attacks such as data theft, data manipulation, or data destruction.

Real-Life scenarios

a) **Digital Watermarking**: Steganography can be used to embed digital watermarks within images or videos, which can help identify the source of the media content and prevent copyright infringement.
b) **Financial Transactions:** Steganography can be used to protect financial transactions by hiding transaction information within images or videos. This can help prevent fraud and theft of financial information.
c) **Journalism:** Journalists can also use steganography to protect their sources and communicate sensitive information without the risk of being exposed. For instance, a journalist may hide confidential information in an image or video file and send it to a trusted colleague, who can then extract the hidden information without arousing suspicion.

2. An image Carnival.png has been provided with this assignment. Use this image as the source image to hide:
After hiding the information, you also need to retrieve the hidden information using Python and OpenCV. (Part 4 done with part 2)
a. Another image of
(i) smaller size,
(ii) same size, and
(iii) larger size,
using Python and OpenCV
Size of the source image = 408 620 3
Size of the smaller image that we have to hide = 208 420 3
Size of the same sized image that we have to hide = 408 620 3
Size of the larger image that we have to hide = 608 820 3

In [1]:
```
import cv2
import numpy as np
import random
```

In [2]:
```
src_img = cv2.imread('Carnival.png')
hidden_img = cv2.imread('hidden.jpg')
```

In [3]:
```
height_src, width_src, channels_src = src_img.shape
print(height_src, width_src, channels_src)
```

```
408 620 3
```

In [4]:
```
height_hidden, width_hidden, channels_hidden = hidden_img.shape
print(height_hidden, width_hidden, channels_hidden)
```

```
200 283 3
```

We first import the required libraries and then read the source image and the image to hide using openCV. The 3rd and 4th block of code sets the height, width and channels of the source image and image to hide and then prints variables.

In [5]:
```
# Encryption function
def encrypt(source,hidden):
    # Check which image is smaller and set the height and width accordingly
    if source.shape[0] < hidden.shape[0]:
        height = source.shape[0]
        width = source.shape[1]
    else:
        height = hidden.shape[0]
        width = hidden.shape[1]

    for i in range(height):
        for j in range(width):
            for l in range(3):

                # v1 and v2 are 8-bit pixel values
                # of source and hidden images respectively
                v1 = format(source[i][j][l], '08b')
                v2 = format(hidden[i][j][l], '08b')

                # Taking 4 MSBs of each image
                v3 = v1[:4] + v2[:4]

                source[i][j][l]= int(v3, 2)

    return source
```

This encrypt function takes two input images, one the source and the second the image to hide and returns the encrypted source image. It then sets height and width, which are used to parse the image pixels, as the height and width of the smaller image of the two.
The nested loops iterates through each pixel in every channel. v1 and v2 are the 8-bit binary representation of the intensity value in the source and hidden image respectively. We take the MSB of the source and hidden image pixel values and concatenate them into a new binary string, v3. We are replacing the LSB of the source image with the MSB of the hidden image. Then we convert the binary string to an integer and update the pixel value of the source image.

```
In [6]: # Decryption function
        def decrypt(encrypted):
            height = encrypted.shape[0]
            width = encrypted.shape[1]

            hidden = np.zeros((height, width, 3), np.uint8)

            for i in range(height):
                for j in range(width):
                    for l in range(3):
                        v1 = format(encrypted[i][j][l], '08b')
                        v1 = v1[4:] + chr(random.randint(0, 1)+48) * 4

                        hidden[i][j][l]= int(v1, 2)

            return hidden
```

This decrypt function takes the encrypted image as input. It then creates an empty image of the same size as the encrypted image. It iterates the image, converts every pixel value to its 8-bit binary representation and extracts the last 4 bits of the encrypted image. It then appends 4 random bits and creates a new binary string with the last 4 bits of the encrypted image and 4 random bits. Finally, it converts the binary string to an integer and stores the pixel at in the empty image.

(i)

smaller size

```
In [7]: hidden_img_small = cv2.resize(hidden_img, (width_src - 200, height_src - 200))

In [8]: height_hidden_small, width_hidden_small, channels_hidden_small = hidden_img_small.shape
        print(height_hidden_small, width_hidden_small, channels_hidden_small)

        208 420 3
```

This snippet of code resizes the image we want to hide so it is smaller than the source image.

encrypt

```
In [9]: encrypted_small = encrypt(src_img, hidden_img_small)
        cv2.imwrite('encrypted_small.png', encrypted_small)

Out[9]: True
```

decrypt

```
In [10]: hidden_small = decrypt(encrypted_small)
         cv2.imwrite('decrypted_small.png', hidden_small)

Out[10]: True
```

These lines of code encrypt and decrypt the smaller hidden image and save them respectively.

(ii)

same size

```
In [11]: hidden_img_same = cv2.resize(hidden_img, (width_src, height_src))

In [12]: height_hidden_same, width_hidden_same, channels_hidden_same = hidden_img_same.shape
         print(height_hidden_same, width_hidden_same, channels_hidden_same)

         408 620 3
```

This snippet of code resizes the image we want to hide so it is the same size as the source image.

encrypt

```
In [13]: encrypted_same = encrypt(src_img, hidden_img_same)
         cv2.imwrite('encrypted_same.png', encrypted_same)

Out[13]: True
```

decrypt

```
In [14]: hidden_same = decrypt(encrypted_same)
         cv2.imwrite('decrypted_same.png', hidden_same)

Out[14]: True
```

These lines of code encrypt and decrypt the same size hidden image and save them respectively.

larger size

```
In [15]: hidden_img_large = cv2.resize(hidden_img, (width_src + 200, height_src + 200))

In [16]: height_hidden_large, width_hidden_large, channels_hidden_large = hidden_img_large.shape
         print(height_hidden_large, width_hidden_large, channels_hidden_large)

         608 820 3
```

(iii)

This snippet of code resizes the image we want to hide so it is larger than the source image.

encrypt

```
In [17]: encrypted_large = encrypt(src_img, hidden_img_large)
         cv2.imwrite('encrypted_large.png', encrypted_large)

Out[17]: True
```

decrypt

```
In [18]: hidden_large = decrypt(encrypted_large)
         cv2.imwrite('decrypted_large.png', hidden_large)

Out[18]: True
```

These lines of code encrypt and decrypt the larger hidden image and save them respectively.

Outputs

(i) Smaller

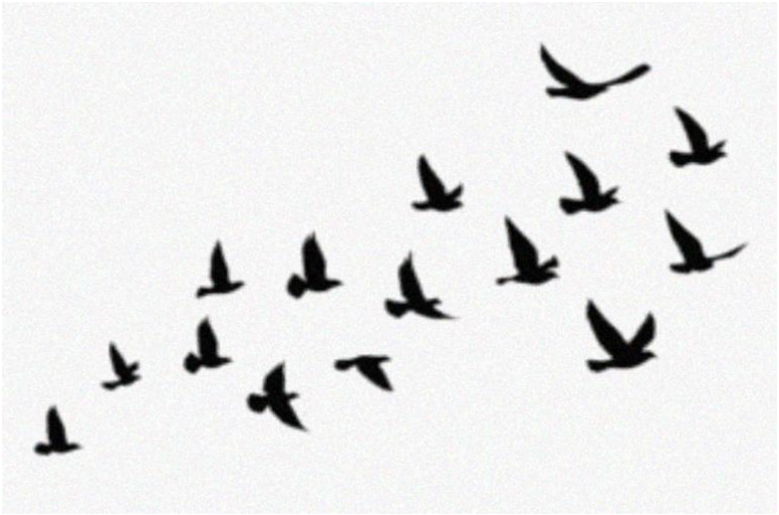
a) Encrypted

b) Decrypted

(ii) Same

a) Encrypted

b) Decrypted



(iii) Larger

a) Encrypted

b) Decrypted

b. Text data comprising of
(i) a word ('word' - size: 4)
(ii) a phrase ('Just breathe' - size: 12)
(iii) a sentence, ('The only way to do great work is to love what you do.' - size: 53)
using Python and OpenCV

In [1]:
```
import cv2
import numpy as np
```

In [2]:
```
# Load the image
img = cv2.imread('Carnival.png')
```

We start by importing the necessary libraries and loading the source image.

In [3]:
```
def hide_text(img, message):
    # convert message to binary string
    binary_message = ''.join(format(ord(c), '08b') for c in message)
    # add end of message delimiter
    binary_message += '00000000'

    # check if message can fit in the image
    height, width, _ = img.shape
    if len(binary_message) > height * width:
        raise ValueError("Message is too large to fit in the image")

    # pad binary message with zeros
    padding = (height * width) - len(binary_message)
    binary_message += '0' * padding

    # convert binary message to numpy array
    binary_array = np.array([int(i) for i in binary_message], dtype=np.uint8)
    # reshape binary array to match image shape
    binary_array = binary_array.reshape(img.shape[:2])

    # replace LSB of each pixel in every channel with message bit
    stego_img = img.copy()
    for i in range(height):
        for j in range(width):
            for k in range(3):
                stego_img[i][j][k] = (stego_img[i][j][k] & 254) | binary_array[i][j]

    return stego_img
```

The hide_text function takes 2 parameters, the source image and the message to hide. It then converts the 'message' string to a binary string using 8 bits per character. It does this by iterating over each character in the 'message' string, getting its ASCII code with the 'ord' function, and converting that code to an 8-bit binary string with the format function. The resulting binary string is concatenated to form the 'binary_message'.

Then we add 8 zeros to the end of the 'binary_message'. This is used as a delimiter to indicate the end of the message.
The next snippet of code gets the height and the width of the image and check whether the message can fit in the image. If it can't, the code raises an error.

We then calculate the number of padding bits needed to fill the remaining space in the image after the message is hidden. The 'padding' variable is assigned the difference between the total number of pixels in the image and the length of the 'binary_message'. Then, padding zeros are added to the end of the 'binary_message'.

```
binary_message = np.array([int(i) for i in binary_message], dtype=np.uint8)
```

This line converts the 'binary_message' string to a numPy array of unsigned 8-bit integers. The int(i) function is used to convert each character in the 'binary_message' string to an integer, and the resulting integers are stored in a list comprehension. The dtype=np.uint8 parameter specifies that the resulting array should have a data type of unsigned 8-bit integers.

```
# convert binary message to numpy array
binary_array = np.array([int(i) for i in binary_message], dtype=np.uint8)
# reshape binary array to match image shape
binary_array = binary_array.reshape(img.shape[:2])
```

These lines convert the binary message into a numpy array of unsigned 8-bit integers and reshape it to match the shape of the input image.

```
# replace LSB of each pixel in every channel with message bit
stego_img = img.copy()
for i in range(height):
    for j in range(width):
        for k in range(3):
            stego_img[i][j][k] = (stego_img[i][j][k] & 254) | binary_array[i][j]

return stego_img
```

First, a copy of the original image is made and stored in 'stego_img' to ensure that the original image is not modified.
Then, a triple nested loop is used to iterate through each pixel in the image, where i and j are the row and column indices of the pixel, and k is the channel index (0 for blue, 1 for green, 2 for red).
For each pixel, the least significant bit (LSB) of each color channel is replaced with a bit from the binary message using the bitwise AND and OR operations. The current pixel value is ANDed with the bit mask 254, which has a binary value of 11111110. This sets the LSB to 0, effectively "erasing" the original value of the LSB. Then, the bit from the binary message is ORed with the result, effectively replacing the erased LSB with the bit from the binary message. This process is repeated for all three color channels.
Finally, the modified image with the hidden message is returned as 'stego_img'.

```
In [11]: def decrypt(img):
             # Flatten the image
             flat_stego = img.reshape((-1, 3))

             # Extract the least significant bit of the blue channel
             binary_msg = np.bitwise_and(flat_stego[:, 0], 1)

             # Convert the binary message to text
             binary_msg = np.packbits(binary_msg)
             message = binary_msg.tobytes().decode('utf-8')
             return message.rstrip('\x00')
```

The decrypt function flattens the input image into a 2D numpy array with three columns (one for each color channel) and as many rows as there are pixels in the image. The '-1' argument to 'reshape' tells numPy to infer the number of rows automatically based on the size of the input array. The next line extracts the least significant bit (LSB) of the blue color channel for each pixel in the flattened image. The 'flat_stego[:, 0]' expression selects the first column of the flattened image (which corresponds to the blue channel), and the '1' argument is a binary mask with a '1' in the LSB position and zeros everywhere else. The 'bitwise_and' function applies this mask to each element of the blue channel, effectively extracting the LSB for each pixel.

```
# Convert the binary message to text
binary_msg = np.packbits(binary_msg)
```

This line converts the binary message (which is a 1D array of 0s and 1s) to a packed binary representation. The 'packbits' function converts each group of 8 bits in the input array into a single byte value.
The next line converts the packed binary message to a UTF-8 encoded string by first calling the 'tobytes' method to convert the packed binary data to a bytes object, and then decoding the bytes using the UTF-8 encoding. The rstrip('\x00') call at the end of the function removes any trailing null bytes from the decoded string.

```
In [12]: word_img = hide_text(img, 'word')
         cv2.imwrite('Carnival_with_word.png', word_img)
Out[12]: True

In [13]: phrase_img = hide_text(img, 'Just breathe')
         cv2.imwrite('Carnival_with_phrase.png', phrase_img)
Out[13]: True

In [15]: sen_img = hide_text(img, 'The only way to do great work is to love what you do.')
         cv2.imwrite('Carnival_with_sen.png', sen_img)
Out[15]: True

In [16]: word_img = cv2.imread('Carnival_with_word.png')
         word = decrypt(word_img)
         print('Decrypted Message:', word)
         Decrypted Message: word

In [17]: phrase_img = cv2.imread('Carnival_with_phrase.png')
         phrase = decrypt(phrase_img)
         print('Decrypted Message:', phrase)
         Decrypted Message: Just breathe

In [18]: sen_img = cv2.imread('Carnival_with_sen.png')
         sen = decrypt(sen_img)
         print('Decrypted Message:', sen)
         Decrypted Message: The only way to do great work is to love what you do.
```

The above snippet of code hides a word, a phrase and a sentence and then retrieves the word, phrase and sentence.

Outputs

(i) Word


```
word_img = cv2.imread('Carnival_with_word.png')
word = decrypt(word_img)
print('Decrypted Message:', word)
```

```
Decrypted Message: word
```

(ii) Phrase


```
phrase_img = cv2.imread('Carnival_with_phrase.png')
phrase = decrypt(phrase_img)
print('Decrypted Message:', phrase)
```

```
Decrypted Message: Just breathe
```

(iii) Sentence


```
sen_img = cv2.imread('Carnival_with_sen.png')
sen = decrypt(sen_img)
print('Decrypted Message:', sen)
```

```
Decrypted Message: The only way to do great work is to love what you do.
```

3. Compare the results of each scenario to see the impact of hidden information size on the source image. When does the source image start to look suspicious?

The visibility of the hidden image after steganography is dependent on various factors such as the size of the source and hidden images, the amount of information in the hidden image, and the steganography technique used. In the case of the given image and steganography technique, when the hidden image is smaller than the source image, it is more visible after steganography. This is because the amount of information that needs to be hidden is less than that in a larger hidden image. As a result, there is less distortion in the source image when hiding a smaller image, making the hidden image more visible. Conversely, when the hidden image is larger than the source image, more information needs to be hidden, leading to more significant distortions in the source image, making the hidden image less visible. Therefore, the size of the hidden image and the amount of information it contains significantly impact the visibility of the hidden image after steganography. Additionally, as more information is hidden, the source image starts to look more suspicious, as there is a greater visible distortion of the image.

When hiding text in the source image, the length of the text can have an impact on the effectiveness of the steganography. A single word is a very small amount of data to hide and is likely to be less noticeable in the image. However, hiding a phrase or a sentence will require more data to be hidden, potentially making the message more apparent in the image. Additionally, the longer the text, the more likely it is that the hidden message will affect the overall image quality and may become more noticeable to someone examining the image closely. Therefore, it is important to consider the length and content of the hidden message when using steganography to ensure that the message remains hidden and the overall image quality is not negatively impacted.

But this is barely visible in the results.

4. After hiding the information, you also need to retrieve the hidden information using Python and OpenCV.

Done with 2

5. Compare the original hidden information with the retrieved information to see the impact of data loss.

When retrieving the hidden images from the steganographic process, the size of the retrieved image is always the same as the size of the source image. This means that if the hidden image is smaller than the source image, the retrieved image will be distorted as it will be stretched to match the dimensions of the source image. The remaining pixels are then filled with the bits of the source image, resulting in the loss of the original hidden information. Similarly, if the hidden image is the same size as the source image, it will also be distorted, and some of the original hidden information may be lost. In the case where the hidden image is larger than the source image, the retrieved image will also be distorted, and significant information loss occurs, as the retrieved image will only contain the portion of the hidden image that fits within the dimensions of the source image. It is, therefore, important to consider the size of the source and hidden images when applying steganography techniques to ensure that the original hidden information is preserved during the retrieval process.

When we hide text in an image using steganography, there is always a possibility of data loss during the retrieval process. However, in general, when retrieving the text, we get back the right text as long as the image has not been significantly altered or corrupted. The retrieved information is compared with the original hidden information to see the impact of data loss. In cases where the hidden information is in the form of text, we can easily compare the retrieved text with the original text to check for any discrepancies or errors. If there is data loss, the retrieved information may be incomplete or may have missing characters or words, which can make it difficult to understand the original message. Therefore, it is important to be aware of the potential for data loss when using steganography to hide information and to take steps to minimize the risk of data loss during the retrieval process.