Image Processing - Project

Fundamentals of LSB Image Steganography

Abhijeet Ambadekar
(16BCE1156)

2018-03-28

Contents

1 Abstract

Steganography is the art of hiding the fact that communication is taking place, by hiding information in other information. The main file formats that are used for steganography are Text, images, audio, video, protocol. This project is developed for hiding information in any image file. For hiding secret information in images, there exists a large variety of steganography techniques some are more complex than others and all of them have respective strong and weak points.

In this project, one such algorithm will be studied and analyzed based on its ability to hide information, the sustainability of information embedded on compression of the images, effect of loss of information on transmission of the image. The algorithm used in this project is the LSB Algorithm (with certain unique modifications) to understand how an image is affected on the use of different position of the bit to inculcated in the algorithm. LSB (Least Significant Bit) substitution is the process of adjusting the least significant bit pixels of the carrier image. It is a simple approach for embedding message into the image.

The software programs used for this project are:

- Python 3
- Python Image Library
- Python File Handling Library

2 Introduction

What is steganography? In a nutshell, the main goal of steganography is to hide information within data that doesn't appear to be secret at a glance. The simplest of its techniques include hiding the message to convey in everyday objects. For example, this sentence:

Since everyone can read, encoding text in neutral sentences is definitely effective

turns into :

Secret inside

if one takes the first letter of every word. Steganography is really handy to use, because people won't even suspect that they're looking at a secret message–making it less likely that they'll want to try to crack someone's code. Steganography takes cryptography a step farther by hiding an encrypted message so that no one suspects it exists. Ideally, anyone scanning one's data will fail to know it contains encrypted data. One use of steganography includes watermarking which hides copyright information within a watermark by overlaying files not easily detected by the naked eye. This prevents fraudulent actions and

gives copyright protected media extra protection.

3 Overview of Image Steganography

Almost all digital file formats can be used for steganography, but the formats that are more suitable are those with a high degree of redundancy. Redundancy can be defined as the bits of an object that provide accuracy far greater than necessary for the object's use and display. The redundant bits of an object are those bits that can be altered without the alteration being detected easily. Image and audio files especially comply with this requirement, while research has also uncovered other file formats that can be used for information hiding.

Given the proliferation of digital images, especially on the Internet, and given the large amount of redundant bits present in the digital representation of an image, images are the most popular cover objects for steganography. To understand more about Image Steganography, one needs to learn more about two specific image properties:

3.1 Image bit depth

To a computer, an image is a collection of numbers that constitute different light intensities in different areas of the image. This numeric representation forms a grid and the individual points are referred to as pixels. Most images on the Internet consists of a rectangular map of the image's pixels (represented as bits) where each pixel is located and its colour. These pixels are displayed horizontally row by row.
The number of bits in a colour scheme, called the bit depth, refers to the number of bits used for each pixel. The smallest bit depth in current colour schemes is 8, meaning that there are 8 bits used to describe the colour of each pixel. Monochrome and greyscale images use 8 bits for each pixel and are able to display 256 different colours or shades of grey. Digital colour images are typically stored in 24-bit files and use the RGB colour model, also known as true colour. All colour variations for the pixels of a 24-bit image are derived from three primary colours: red, green and blue, and each primary colour is represented by 8 bits. Thus in one given pixel, there can be 256 different quantities of red, green and blue, adding up to more than 16-million combinations, resulting in more than 16-million colours. Not surprisingly the larger amount of colours that can be displayed, the larger the file size.

3.2 Image compression

When working with larger images of greater bit depth, the images tend to become too large to transmit over a standard Internet connection. In order to display an image in a reasonable amount of time, techniques must be incorporated to reduce the image's file size. These techniques make use of mathematical formulas to analyze and condense image data, resulting in smaller file sizes. This process is called compression. In images there are two types of compression: lossy and lossless. Both methods save storage space, but the procedures that they implement differ.

Lossy compression creates smaller files by discarding excess image data from the original image. It removes details that are too small for the human eye to differentiate, resulting in close approximations of the original image, although not an exact duplicate. An example of an image format that uses this compression technique is JPEG (Joint Photographic Experts Group).

Lossless compression, on the other hand, never removes any information from the original image, but instead represents data in mathematical formulas. The original image's integrity is maintained and the decompressed image output is bit-by-bit identical to the original image input. The most popular image formats that use lossless compression is GIF (Graphical Interchange Format) and 8-bit BMP (a Microsoft Windows bitmap file).

Compression plays a very important role in choosing which steganographic algorithm to use. Lossy compression techniques result in smaller image file sizes, but it increases the possibility that the embedded message may be partly lost due to the fact that excess image data will be removed. Lossless compression though, keeps the original digital image intact without the chance of lost, although is does not compress the image to such a small file size.

4 Least Significant Bit Image Steganography

4.1 Value of LSB

In a gray scale image each pixel is represented in 8 bits. The last bit in a pixel is called as Least Significant bit as its value will affect the pixel value only by "1". So, this property is used to hide the data in the image. If anyone have considered last two bits as LSB bits as they will affect the pixel value only by "3". This helps in storing extra data. The Least Significant Bit (LSB) steganography is one such technique in which least significant bit of the image is replaced with data bit. As this method is vulnerable to steganalysis so as to make it more secure we encrypt the raw data before embedding it in the image.

In other words, the least significant bit (in other words, the 8th bit) of some or all of the bytes inside an image is changed to a bit of the secret message. Digital images are mainly of two types: (i) 24 bit images and (ii) 8 bit images. In 24 bit images we can embed three bits of information in each pixel, one in each LSB position of the three eight bit values. Increasing or decreasing the value by changing the LSB does not change the appearance of the image, much so the resultant output image looks almost same as the input image. In 8 bit images, one bit of information can be hidden.

So why the interest in the least significant bit (LSB) of a number. Taking the LSB of a number is beneficial in comparison to any other bit because the change caused by changing the LSB causes the least possible difference that can occur in the value of the pixel. Such insignificant changes are almost invisible to human vision and it is very difficult to distinguish two pixel having their values differ by one.

For example, 139 in binary can be written as **10001011**. Changing the LSB gives us the new binary value **10001010** which in decimal system is 138. However, changing the MSB of the image returns **00001011** which is just 11 which is far less from the starting value.

4.2 LSB Embedding

Least Significant Bit (LSB) embedding is a simple strategy to implement steganography. Like all steganographic methods, it embeds the data into the cover so that it cannot be detected by a casual observer. The technique works by replacing some of the information in a given pixel with information from the data in the image. While it is possible to embed data into an image on any bit-plane, LSB embedding is performed on the least significant bit(s). This minimizes the variation in colors that the embedding creates.

For example, embedding into the least significant bit changes the color value by one. Embedding into the second bit-plane can change the color value by 2. If embedding is performed on the least significant two pixels, the result is that a color in the cover can be any of four colors after embedding. Steganography avoids introducing as much variation as possible, to minimize the likelihood of detection. In a LSB embedding, we always lose some information from the cover image. This is an effect of embedding directly into a pixel. To do this we must discard some of the cover's information and replace it with information from the data to hide. LSB algorithms have a choice about how they embed that data to hide. They can embed losslessly, preserving all information about the data, or the data may be generalized so that it takes up less space.

5 Encoding an Image with Text

5.1 Module descrption

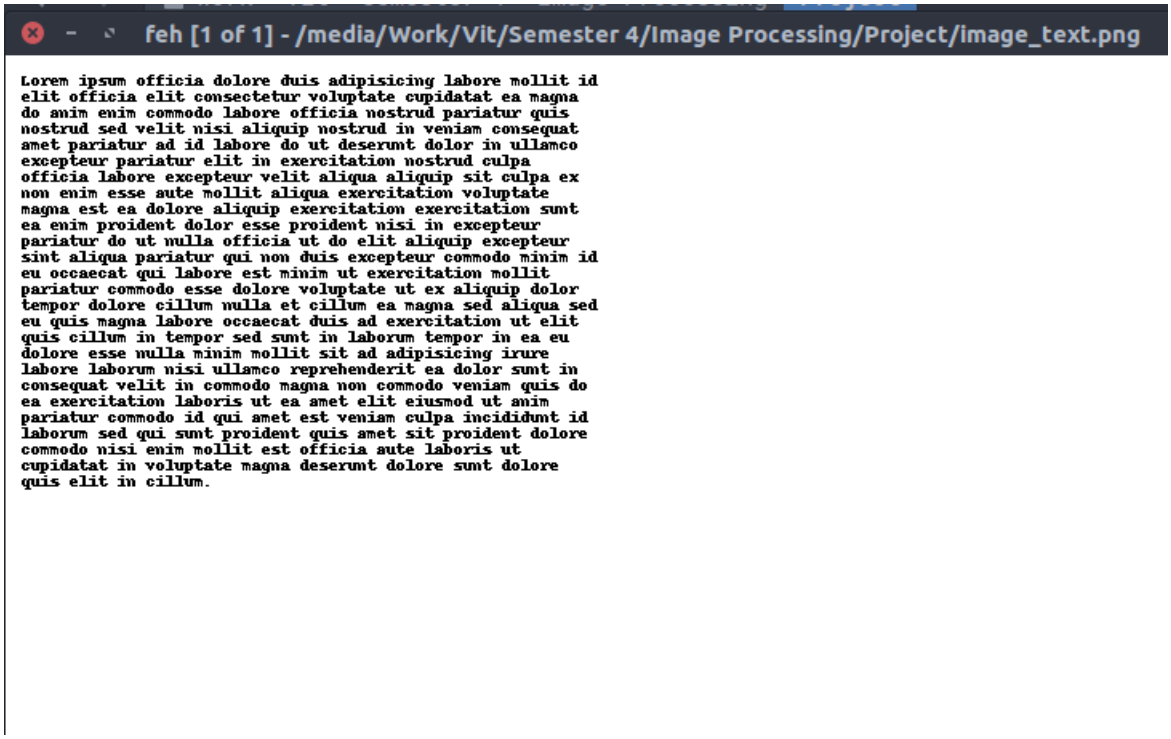
The encoding procedure is a two part process:
First, we create an image with the same dimensions as the input image (to be used for embedding) with the text to be embedded being written in the image. This is done by the Text_to_Image module.
Second, the image obtained from the Text_to_Image module is masked over the input image to obtain the embedded (or "stego") image. Both the stages are explained further.

5.2 Getting familiar with Text to Image module

This module will take in the input text and the dimensions of the input image and create a grayscale image of the given dimensions with the text written in the form of pixels. An implementation of this module is listed below:

```
def write_text(text_to_write, image_size):
        image_text = Image.new("RGB", image_size)
        font = ImageFont.load_default().font
        drawer = ImageDraw.Draw(image_text)

        margin = offset = 10

        for line in textwrap.wrap(text_to_write, width=60):
                drawer.text((margin,offset), line, font=font)
                offset += 10
        return image_text
```

For a given text given to this module, say in the form of `write_text("Ullamco cillum deserunt.... Some random gibberish text")`, the output it will produce is:



5.3 Encoding Module - with Examples

The masking process mentioned earlier is a simple process. The algorithm is as follows:

- If pixel value of image obtained from Text_to_Image module is Low (or its colour is Black), make the LSB of the red channel value of the corresponding pixel of the input image as '0'.
- Else, make the LSB of the red channel value of the corresponding pixel of the input image as '1'.

What this process has acheived is that the image is masked over the LSB red channel value of the whole input image. Thus the message has been hidden inside the input image and the obtained "Stego" image is visually indistinguishable from the input image. The code for the encoding module is given as follows:

```
def encode_image(text_to_encode, template_image_name="image.jpg", bit_to_encode=-1):
    template_image = Image.open(template_image_name)
    red_template = template_image.split()[0]
    green_template = template_image.split()[1]
    blue_template = template_image.split()[2]

    x_size = template_image.size[0]
    y_size = template_image.size[1]

    image_text = write_text(text_to_encode, template_image.size)
    bw_encode = image_text.convert('1')

    encoded_image = Image.new("RGB", (x_size, y_size))
    pixels = encoded_image.load()
    for i in range(x_size):
        for j in range(y_size):
            red_template_pix = format(red_template.getpixel((i,j)), '#010b')[2:]
            old_pix = red_template.getpixel((i,j))
            tencode_pix = bin(bw_encode.getpixel((i,j)))

            if tencode_pix[-1] == '1':
                # red_template_pix = '1' + red_template_pix[1:]
                temp = list(red_template_pix)
                temp[bit_to_encode] = '1'
                red_template_pix = "".join(temp)
            else:
                # red_template_pix = '0' + red_template_pix[:-1]
                temp = list(red_template_pix)
                temp[bit_to_encode] = '0'
                red_template_pix = "".join(temp)
            pixels[i, j] = (int(red_template_pix, 2), green_template.getpixel((i,j)), blue_template.getpixel((i,j)))
    new_img_name = template_image_name.split('.')[0] + "_enc.png"
    encoded_image.save(new_img_name)
    return new_img_name
```

What can be observed is that this Python function is generic when it comes to the bit to which it has to encode the value of the text. This provides us with an insight of the how embedding the text in different bits of the pixel values will affect the image in terms of visual perception. For example the input text used in the previous section will be embedded into the input image of a fruit basket at different bit positions to describe the visual characterstics of the image:

Figure 1: Input image of a Fruit Basket

Figure 2: Stego Image obtained after MSB Steganography

Figure 3: Stego Image obtained after Steganography on 2nd bit from the left

Figure 4: Stego Image obtained after LSB Steganography

6 Decoding Module

6.1 Module Description

Because of the sophistications and the complexity of the encoding module have been thoroughly implemented, it leave s very little work for the decoding module. Its algorithm is as follows:

- If the LSB of the red channel value of a pixel of the Stego image is '1' or HIGH, make the complete pixel value as (255,255,255) or in other words make the pixel colour as "White".
- Else, make the complete pixel value as (0,0,0) or in other words make the pixel colour as "Black".

6.2 Decoding module usage - with Example

The code for the decoding module is given as:

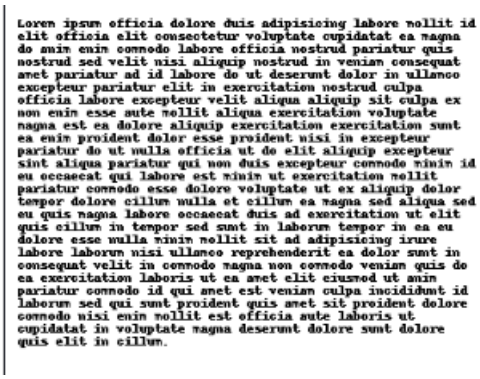
```
def decode_image(file_location="image_enc.png", bit_to_decode=-1):
    encoded_image = Image.open(file_location)
    red_channel = encoded_image.split()[0]

    x_size = encoded_image.size[0]
    y_size = encoded_image.size[1]

    decoded_image = Image.new("RGB", encoded_image.size)
    pixels = decoded_image.load()

    for i in range(x_size):
        for j in range(y_size):
            if format(red_channel.getpixel((i, j)), '#010b')[2:][bit_to_decode] == '0':
                pixels[i, j] = (255, 255, 255)
            else:
                pixels[i, j] = (0,0,0)

    new_img_name = file_location.split("_enc")[0] + "_text.png"
    decoded_image.save(new_img_name)
    return new_img_name
```

Similar to the encoding module, the decoding module is also generic with respect to the bit position to decode. If the person knows which bit position has been used to embedded the text in the stego image, one can easily recover the text from the stego image. For example the text recovered from the Image obtained from the LSB Steganographic embedding of the input text is given as:

Figure 5: Input image of a Fruit Basket

Figure 6: Stego Image obtained after LSB Steganography

Figure 7: Text recovered form Stego Image



7 Future Work

Encryption can be powerful technique combined with Steganography. The bit encrypted using a suitable Asymmetric key cryptography system will make the embedded text completely undetectable.
There is also another possibility of exploring various patterns in the LSB's of the 3 different pixel values of RGB representation of a pixel. More random patterns make the text more difficult to recover unless the methodology of embedding is fully known.

8 Results and Conclusion

The reversible perturbation of values used in steganography enables the embedding of data into a cover medium. Choosing to modify values that have a small affect on the cover medium limits the ability to detect the embedding. Embedding strategies may be easily derived and implemented to complicate detection and inhibit the retrieval of the message by a third party, while still allowing easy retrieval by the intended recipient. LSB Embeddings may be detected simply through visual inspection of an image and its bit-planes, or more reliably through methods which use statistical metrics to identify the likelihood an image contains hidden data. While an embedding may be detected, it may not be easily decoded, nor may a stego object be discovered due to the sheer number of images available. Steganography proves to be a significant technique for evading detection when communicating. The detection issues with steganography create challenges for security systems in attempting to prevent the transmission of steganographic content. As the need to communicate in secret will always exist, steganography will likely continue to play an important role enabling covert communication.

9 References

1. Fridrich, J., Goljan, M., & Du, R. (2001). Reliable detection of LSB steganography in color and grayscale images. Proceedings of the 2001 workshop on Multimedia and security new challenges - MM&Sec '01, 27. New York, New York, USA: ACM Press. doi:10.1145/1232454.1232466

2. `http://effbot.org/imagingbook/image.htm#image-open-function` For the study of Python Image Library.

3. `http://ijact.org/volume3issue4/IJ0340004.pdf`