CS808 Computer Security Fundamentals
Coursework Assignment

Group A

Academic Year 2021/2022

Table of Contents

The Task

Programming Route: Requirements

- Hide text file in 24bit bitmap cover image using LSB
- Asks user to provide name of text file
- Ensure sufficient capacity to hide payload
- Extract said payload-containing text file from bitmap
- Written in **Python**
- Code must be tested on as many machines / OS's as possbile
- Possible errors identified
- PEP 8 standard (clear commenting, snake case, meaningful variables, etc.)
- Header information at the start of files (estimated image size, height, width, etc.) See here

Programming Route: Constraints

- Cover image should be 24bit .bmp
- Assume a portion of LSB's of the cover image are required at the start to account for the size of the payload
- First 54bytes of image not altered
- No external Python libraries but, can use Pillow Library to work with images
- Will be tested on Windows 10

Video Presentation

1. Overview of steganography and LBS algorithm (1 minute)
2. Feasability of developing this data smuggling software and, demonstration of working software (3 minutes)
3. Wider impact of scenario, i.e. insider threat risk in steganography data smuggling (3 minutes)
4. Recommendations to manage risk of such brief (3 minutes)

Submission

- Submit: [1] video presentation, [2] code (.py file), [3] code instructions / running instructions / executive report

2 and [3] should be compressed into a .zip file before upload

- We will choose to include a plain text (Markdown (.md)) version, TeX document typeset PDF, and Beamer slide show PDF
- One submission per team, by nominated member
- A **workload record** must also be submitted, detailing the effort of each team member and their relative contribution to the coursework. Score 1→5 accordingly. If nothing submitted, assumed that each contributed equally. Inclide in .zip
- Complete **peer assessment** on MyPlace

1 Steganography & LSB Overview

1.1 Steganography Basics

Steganography is a form of data covering favoured by participants who wish to hide shared information 'in plain sight', through a very standard looking method such as multimedia transfer. Steganography (*Steganos* / *Graphie*) quite literally means 'covered writing' from its Greek derivation. It's becoming particularly popular due to the current generation of business machines being mobile-' and hand-held-device-based which often have seemless access to features required (Pereira, Sousa, 2004).

The steganography process involves three critical components: [1] the 'cover object', which is the object (the multimedia, such as an image) that you wish to hide data in; [2] the 'payload', which is the data (object) you wish to hide and; [3] the 'stego-object', which is the altered version of the *cover object* that now contains the *payload*. It's said that the human eye should be indifferent to sight of *stego-object* and the original *cover object* to best disguise the *payload*. That is for example, when using the Least Significant Bit (LSB) approach, an edited pixel of an image should be so insignificant that it looks identical to the original, to a human eye (Singh, et. al., 2015). Unlike cryptography, steganography doesn't hide the existence of a message (*payload*), it disguises it in already existing content. This could be argued to decrease suspicion surrounding message secracy. Beacuse this form of security uses already existing content, it's also very cheap and accessible to users in-the-know.

1.2 Least Significant Bit (LSB) Approach

The general steganography framework of multimedia embeddedness covers transfer in the form of [1] text, [2] imagery, [3] video, [4] audio and, [5] networks. Which, when using the LSB approach, all involve generally the same technique of reconfigurating data to minorly alter the properties of multimedia. Using imagery (a digital uncompressed image) as an example in the context of LSB, alteration is commonly made to the final bits in 8bit (byte) binary sequences of pixels to embed a message. The final pixel is known as the 'least significant bit' as altering it has the smallest effect on the visual appearance of the image post-*payload*-addition. That is, the *stego-object* is not distinguishable from the *cover obect*, to the human eye. This is why high-contrast, color-varying images are most effective for this; larger changes are less noticable when

compared to more monotone images. LSB is regarded one of the most versatile and important applications of steganography today, due to its application in RGB bitmaps, and JPEG attribute frequencies, etc. (Singh, et. al, 2015).

1.3 In Practice

Pre-Payload	Post-Payload
01010010	01010011
01001010	01001010
10010111	10010111
11001100	11001101
11010101	11010100
01010111	01010111
00100110	00100110
01000011	01000011

Table 1.1: LSB Example

LBS is most easily understood using an $N \times M$ grey-scale image (Picione, et. al., 2006). This is because each pixel can be represented as in 8bit binary $\forall\{0 \longrightarrow 255\}$. For example, to embed the letter 'Z' in a sample of eight pixels from an image, as Z's binary representation is 8bit, the modifications are made to the pixels as seen in Table 1 (Singh, et. al., 2015).

2 The Program & Application Feasability

This program is designed to request an image (*cover object*), request a payload-bearing plain text (.txt) file (*payload*), and embed the content of the text file in the given image to produce a new image which is unnoticeably different to the original image, to the human eye (*stego-object*). The program is also designed to request an image (*stego-object*) and extract embedded text content (*payload*) to display upon output.

2.1 Functional Requirements

In summary, this program performs and allows a user to do the following:

- Encode a message in an image
- Decode a message from an image
- Requests the name of a plain text (.txt) file
- Reads, extracts and stores the text within the text file
- Embeds the stored text into an image of the user's choice

2.2 Engineering Requirements

This program meets the required prgramming standards, such that it:

- Is written in Python
- Uses only Pillow 8.4.0 – PIL (Pillow Image Library) Fork externally
- Is interpreted and tested using python3
- Uses a 24bit bitmap (.bmp) as a sample to encode upon
- Checks the target image size against the target text file size to ensure there is enough space to store the text in the image
- Identifies any relevant Errors and returns user with options
- Fulfils PEP 8 standards, including commenting, letter-casing, and variables, etc.
- Tested on various machines and operating systems

2.3 Engineering

1. Module dependencies: Makes use of Pillow 8.4.0 – PIL Fork; PIL (Python Image Library) is a fork allowing mod of image pixels

```
from PIL import Image
```

2. Function: data to 8bit binary from ASCII

```
def conv_bin(data):
    data_bin = []
    for n in data:
        data_bin.append(format(ord(n), "08b"))
    return data_bin
```

3. Function: convert image pixels relative to binary generated

```
def conv_pixl(pixl, data):
    data_comp = conv_bin(data)
    data_length = len(data_comp)
    data_img = iter(pixl)

    for d in range(data_length):
        pixl = [
            value for value in data_img.__next__()[0:3] +
            data_img.__next__()[0:3] +
            data_img.__next__()[0:3]
        ]
        for b in range(0, 8):
            if (data_comp[d][b] == "0" and pixl[b] % 2 != 0):
                pixl[b] -= 1
            elif (data_comp[d][b] == "1" and pixl[b] % 2 == 0):
                if(pixl[b] != 0):
                    pixl[b] -= 1
        if (d == data_length - 1):
            if (pixl[-1] % 2 == 0):
                if(pixl[-1] != 0):
                    pixl[-1] -= 1
                else:
                    pixl[-1] += 1
        else:
            if (pixl[-1] % 2 != 0):
                pixl[-1] -= 1
```

```
            pixl = tuple(pixl)
            yield pixl[0:3]
            yield pixl[3:6]
            yield pixl[6:9]
```

4. Function: new pixels into an image

```
def inp_pixel(img_gen, data):
    p = img_gen.size[0]
    (x, y) = (0, 0)
    for pixel in conv_pixl(img_gen.getdata(), data):
        img_gen.putpixel((x, y), pixel)
        if (x == p - 1):
            x = 0
            y += 1
        else:
            x += 1
```

5. Function: read data from file and store contents in variable

```
def txt_read(data):
    inp_file = open(data, "r")
    content = inp_file.read()
    inp_file.close
    return content
```

6. Function: encode data from desired message into newly generated image

```
def encode():
    img = input(
        "\nType the name of the existing image w/ extensio"
        "n (e.g. vimtutor.png)\nthen;\nPress 'Enter' to exe"
        "cute\n>> "
    )
    image = Image.open(img, "r")
    data = txt_read(
        input(
            "\nType the name of the document containing text t"
            "o be encoded\nthen;\nPress 'Enter' to Execute\n>> "
        )
    )
    (w, h) = img_cont.size
    img_size = h * w * 24
```

```
    if len(data) > img_size:
        raise RuntimeError(
            "\nFATAL ERROR, FOOL! Text file too big! Try s"
            "toring a smaller text file, unde"
            "r " + str((img_size / 8) /1024) + "kB!"
        )
    if (len(data) == 0):
        raise ValueError("\nFATAL ERROR, FOOL! No location declared! Try again!")
    img_gen = image.copy()
    inp_pixel(img_gen, data)
    img_gen_title = input(
        "\nType the desired name of the generated im"
        "age w/ desired extension (e.g. neovimtutor."
        "png)\nthen;\nPress ‘Enter’ to execute\n>> "
    )
    img_gen.save(img_gen_title, str(img_gen_title.split(".")[1].upper()))
```

7. Function: decode data from message in target image

```
def decode():
    img = input(
        "\nType the name of the existing image w/ ex"
        "tension (e.g. neovimtutor.png)\nthen;\nPress "
        "‘Enter’ to execute\n>> "
    )
    image = Image.open(img, "r")
    data = ""
    img_data = iter(image.getdata())
    while (True):
        pixels = [
            value for value in img_data.__next__()[0:3] +
            img_data.__next__()[0:3] +
            img_data.__next__()[0:3]
        ]
        search = ""
        for b in pixels[0:8]:
            if (b % 2 == 0):
                search += "0"
            else:
                search += "1"
        data += chr(int(search, 2))
        if (pixels[-1] % 2 != 0):
```

```
                return data
```

8. Function: interfaceable function

```
def interface():
    print(
        "================================================"
        "======\n== Welcome to the Steganography Tutor — "
        "Version 1.7 ==\n================================"
        "======================\n"
    )
    a = input(
        "Type ‘E’ to ENCODE a message\nor;\nType ‘D’ to DE"
        "CODE a message\nthen;\nPress ‘Enter’ to execute\n>> "
    )
    if (a == "E"):
        encode()
    elif (a == "D"):
        print("\nDecoded message follows:\n<< " + decode())
    else:
        raise Exception("\nChoose a valid option")
```

9. Driver Code Program: testing functionality of interface

```
if __name__ == "__main__":
    interface()
```

10. PEP 8 Standard test using `pycodestyle`. Output follows

Yeah, nothing beacuse it follows the PEP 8 standard.

2.4 Interfacing

0.1. Files required

```
[12:46:10 21-10-27] lewisb ~/Documents/Masters/CS808/Assignment $ ls -l
steg_big.txt
steg.bmp
steg_empty.txt
steg_finance.txt
steg.py
steg.txt
```

0.2. Running with `python3` interpreter

```
[12:46:12 21-10-27] lewisb ~/Documents/Masters/CS808/Assignment $ python3 steg.py
```

1.1. First promt: encode or decode. Here, enter 'E' or 'D' respectively

```
=======================================================
== Welcome to the Steganography Tutor - Version 1.7 ==
=======================================================

Type 'E' to ENCODE a message
or;
Type 'D' to DECODE a message
then;
Press 'Enter' to execute
>>
```

1.1.1. Exception catch if user enters an invalid character

```
Exception:
Choose a valid option
```

2.1. Second prompt (if user elects to encode). Here, enter image name given in files required `steg.bmp`

```
Type the name of the existing image w/ extension (e.g. vimtutor.png)
then;
Press 'Enter' to execute
>>
```

2.1.1. RuntimeError catch if the user selects a text file too large to be stored in the image (larger than the image). Where 262.79kB is the size of the sample over-sized `steg_big.txt` file

```
RuntimeError:
FATAL ERROR, FOOL! Text file too big! Try storing a smaller text file,
under 262.79296875kB!
```

2.1.2. ValueError catch if the user selects a text file with no text. Sampled with `steg_empty.txt`

```
ValueError:
FATAL ERROR, FOOL! No data present! Try again!
```

2.2. Second prompt (if user elects to decode). Here, enter image name (assuming prior creation using step 4) `neosteg.bmp`

```
Type the name of the existing image w/ extension (e.g. neovimtutor.png)
then;
Press ‘Enter’ to execute
>>
```

3.1 Third prompt (if user elects to encode). Here, enter text file name given in files required `steg.txt`

```
Type the name of the document containing text to-be encoded
then;
Press ‘Enter’ to Execute
>>
```

3.2 Third display (if user elects to decode)

```
<contents of steg.txt here>
```

4. Fourth promt (if user elects to encode). Here, enter a valid name such as `neosteg.bmp` to store the new image

```
Type the desired name of the generated image w/ desired extension (e.g. neovimtutor.png)
then;
Press ‘Enter’ to execute
>>
```

2.5 Testing & Error Identification

Operating System	Architecture	Interpreter	Functional?	Issues Recognized
Arch Linux	Unix	python3	Y	N/A
Manjaro Linux	Unix	python3	Y	N/A
MacOS X	Unix	python3	Y	N/A
Windows XP Professional	DOS	python	N	No PIL support
Windows 7 Home	DOS	python2 (2.7)	Y	Using old PIL version (fix)
Windows 10 Home	DOS	python3	Y	N/A

Table 2.2: Program Testing

2.5.1 Successful Result

(i) Cover Object (Pre-Payload) (ii) Stego-Object (Containing Payload)

Figure 2.1: Successful Payload Encoding

Figure 2.1 shows successfull encoding of a payload within a valid cover object as changes are unnoticeable to the human eye, at a glance. Note that the images contained in Sub-figures (i) and (ii) of Figure 2.1, which highlight payload encoding, are PNG versions of the BMP images used in practice. This is the case as TEX does not support the Microsoft Bitmap image format. Process follows:

Cover Object: `steg.bmp`

$\longrightarrow$

Payload: `steg.txt`

$\longrightarrow$

Stego-Object: `neosteg.bmp`

2.5.2 Successful Result in Context

(i) Cover Object (Pre-Payload) (ii) Stego-Object (Containing Payload)

Figure 2.2: Successful Payload Encoding in Context

We must recall the context of this scenario however. It is unlikely that a financial company will simply be sharing a text document with information about a local Giffnock upholsterer in it. Although, it is also unlikely that a financial company will be suing such a basic small-scale program to transfer sensitive data. They would most likely use a large-scale and more secure version of this program to store CSV files. Therefore, the example shown in Figure 2.2 uses the same `steg.bmp` image as the cover object however, it embeds a slightly larger text file with details which would be present in a typical stock broker's spreadsheets but, for just one client with 23 stocks; thus, making the details printable in a purely vertical plain text file. The stego-object in Figure 2.2 highlights the fact that this solution is still viable, and on a relatively larger scale, would work in the given context. Once again, the images contained in Subfigures (i) and (ii) of Figure 2.2 are the PNG versions of the BMP images used. Porcess follows:

Cover Object: `steg.bmp`

$\longrightarrow$

Payload: `steg_finance.txt`

$\longrightarrow$

Stego-Object: `neosteg2.bmp`

2.5.3 Unsuccessful Result (Text File Too Large)

As discussed in step 2.1.1. of Section 2, Subsection 4, when the user inputs the name of a desired payload-containing text file which is too big to be stored in the targe cover image, an error is returned and the user is prompted to start again, with an appropriate text file. The relevant process of this follows:

Cover Object: `steg.bmp`

$\longrightarrow$

Payload: `steg_big.txt`

$\longrightarrow$

Stego-Object: `<no output>`

2.5.4 Unsuccessful Result (Text File Empty)

As discussed in step 2.1.2. of Section 2, Subsection 4, when the user imput the name of a desired payload-containing (or so they think) text file which actually contains no text, an error is returned and the user is prompted to start again, with an appropriate text file. The relevant process of this follows:

Cover Object: `steg.bmp`

$\longrightarrow$

Payload: `steg_empty.txt`

$\longrightarrow$

Stego-Object: `<no output>`

2.6 Reflection & Final Notes

The only major change to thi program which would be useful is utilization of Unix command line syntax. That is, using operators, functions and flags as opposed to input prompts, meaning only one line has to be inputted by the user. Much like most command line-based programs. The functionality already exists in the program, it would just be an extra hour's worth of work to get it going. Something I'll take care of in my spare time.

3 Wider Impact

See video.

4 Risk Management & Recommendations

See video.

Bibliography

[1] Pereira, V., Sousa, T. (2004). *Evolution of Mobile Communications: from 1G to 4G.* International Working Conference on Performance Modeling and Evaluation of Heterogeneous Networks

[2] Picione, D., Battisti, F., Carli, M., Astola, J., Egiazarian, K. (2006). *A Fibonacci LSB Data Hiding Technique.* 14th European Signal Processing Conference, EURASIP

[3] Singh, A., Singh, J., Singh, H. (2015). *Steganography in Images Using LSB Technique.* International Journal of Latest Trends in Engineering and Technology