Abertay University

LSB Image Steganography Tool Creation

Ance Strazdina

Introduction to Security – CMP110

BSc Ethical Hacking Year 1

2020/21

.

Abstract

Steganography is an ancient practice that has evolved and still retains its functionality today. With the popularity increase of steganography in a digital setting, it is important to be familiar with this technique and methods relating to it. This project discusses steganography as a whole – its history, uses, and working fundamentals and takes a closer look at LSB (least significant bit) steganography in images. By gathering information about this type of steganography, including the way it operates and how others have approached it, an LSB image steganography tool was created using Python. The resulting product is a working steganography tool that gives insight into the basics of LSB steganography and why it is important.

.

Contents

.

1 INTRODUCTION

1.1 BACKGROUND

Steganography is the practice of hiding secret data within something seemingly ordinary and non-secret. The main reason for using it is to conceal the very existence of the hidden data therefore not attracting unwanted attention to it and enabling secrecy (Semilof, Clark, 2018; Cheddad et al., 2010). The term originates from the Greek words steganos meaning hidden or covered and the root graph which means to write (Semilof, Clark, 2018).

The earliest use of this practice dates to 440 BC and it has been continuously used and evolved throughout history. Its earliest uses were in physical form such as using invisible ink (Siper, Farley, Lombardo, 2005). During wartime, members of the espionage used knitting to hide secret messages in clothing (Zarelli, 2017).

In the present day, steganography is still widely used and with the evolution of technology new information hiding techniques have been developed - it is now being used in digital environments (Borse, Anand, Patel, 2013). This way of utilizing steganography is now more widespread than using it physically. Ira Winkler, lead security principal at Trustwave, defines steganography as the hiding of one file within another (Dickson, 2020). The use of this changed definition reveals just how common digital steganography is.

Digital steganography involves hiding data also referred to as payload in various multimedia carriers (stego-containers) such as image, audio, and video files (Rupali, 2020). With the right tools, anyone can use it. Multiple software applications such as Xiao Steganography and Steghide are available for use to perform steganography (Shankdhar, 2020). Same way tools such as StegExpose and StegAlyze can be used to determine if steganography has been used on files by performing steganalysis – the practice of detecting steganography, by finding hidden data and anomalies in files (Dickson, 2020).

In practice, some human rights activists and dissidents use steganography to send sensitive information and it is still used for communication among spies (Dickson, 2020); however, it has been increasingly used to deliver cyber-attacks (Shulmin, Krylova, 2017). Cybercriminals use this practice to hide malicious payloads and scripts in carriers such as image and audio files (Dickson, 2020). Other carriers used for this are Windows applications such as Excel or Word. By opening a file that has been modified this way, the victim releases a hidden script which then installs an application capable of infecting the victim's machine with malware (Stanger, 2020). An instance of this method being used is for Snatch ransomware which first appeared in late 2018. It rebooted the infected machines into safe mode to avoid detection by antivirus software, encrypted the user's files, and then demanded a sum between 1 and 5 Bitcoin for the decryption key (Data Recovery Specialists, 2019). Cases like this are why steganography is relevant to cybersecurity – it is important to be familiar with these methods to ensure attacks like this are prevented more efficiently.

Another use for steganography in security is coupling it with other techniques such as watermarking. A watermark is a signal embedded into digital data that can be used afterward to confirm the legitimacy of the data. It is hidden in the carrier in a way that it cannot be removed without damaging the carrier. This keeps the data accessible all while keeping intellectual property safe (Lu, 2005). Steganography is also used together with cryptography. While encrypted data is bound to attract attention, hiding it in a stego-container solves this problem. On top of that, if steganography fails and the encrypted data is discovered, it still needs to be decrypted (El-Emam, 2007). Figure 1-1 (Cheddad, 2009, p. 7) demonstrates some of the methods used in security which are often coupled with steganography.

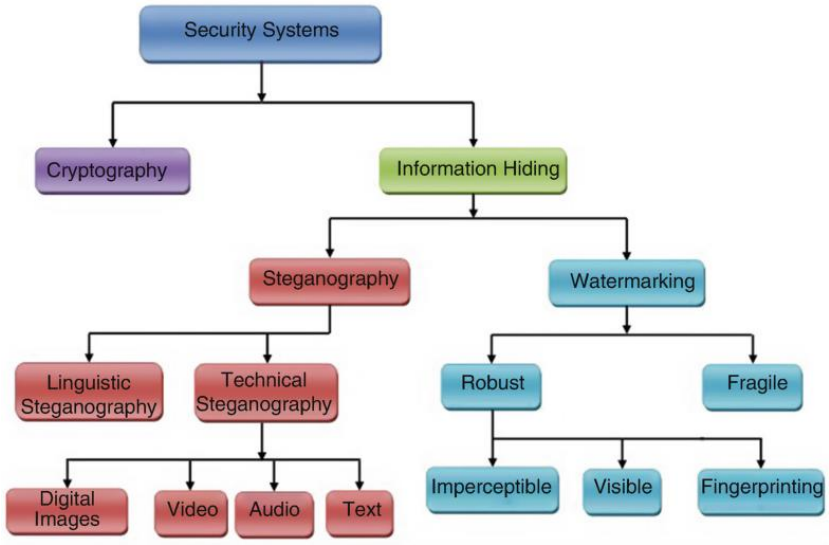

Figure 1-1

While all available file formats can be used for hiding data, files with more redundancy are most suited for this (Borse, Anand, Patel, 2013). Redundant information is unneeded and duplicate information, in this case, it refers to redundant bits which are extra binary digits that are generated and moved with a data transfer to ensure that no bits are lost during the data transfer (Posey, 2015). These extra bits ensure that the file can be altered without the alteration being detected as easily. For this reason, image and audio files are popular carriers as both meet these requirements but considering the wide use of digital images, especially on the internet, and taking the large number of redundant bits in a digital image into account, images are the most popular stego-containers (Borse, Anand, Patel, 2013).

This project focuses on LSB (least significant bit) steganography in images. Although poor in security, this method allows to quickly embed and extract information and is easy to implement. On top of that, it has a high hiding capacity (Wu, 2015). This method works by hiding information in the least significant bits of the carrier and the smaller the payload is, the less apparent is the fact that steganography has been used on a file (Shulmin, Krylova, 2017). In images, pixels hold values that represent colours and the LSB technique replaces the least significant bits of these values with data bits of the message it is trying to hide (Rupali, 2020).

Figure 1-2 (Cortes, 2019) demonstrates the working fundamentals of the LSB technique in images further. Each pixel contains red, green, and blue values which are represented by a number. Here these numbers are displayed in binary format. In this example, the first 3 bits ‘011’, from the full message (the

word ‘ledger’ in binary) seen at the top of the image are encoded in a single pixel. As the least significant bits of the three colour values combined are ‘001’ only the second of the least significant bits is changed to make ‘011’. The result changes the colour of the pixel slightly, but it is impossible for the naked eye to notice. This process is then repeated on the next pixel and so on until the entire message is hidden. To increase hiding capacity, more than one least significant bit can be used; however, the more of these bits are used, the more apparent it is that steganography has been applied to the image which defeats the purpose of using it.

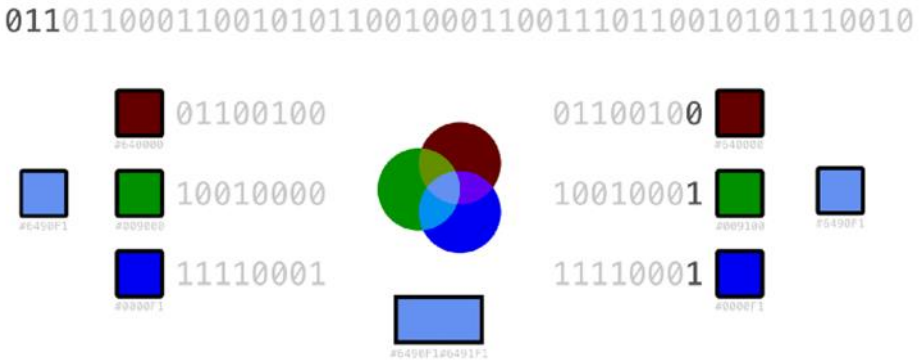

Figure 1-2

When working with images, their file types can be divided into three classes – those with lossy compression, those with lossless compression, and raw file types. Lossy and lossless are used more commonly because of their compact size, and while raw file types are the easiest to work with, they tend to be very large (Scott, 2018). Lossy image files such as JPEG, tend to get modified slightly when subjected to storage, transmission, or decompression. These changes are virtually indistinguishable; however, because of this performing LSB steganography on lossy files is trickier than using lossless files because the encoded message might get lost. Lossless files are the opposite of lossy files, no data is changed when performing actions on the image, so these files are favourable for LSB steganography. Examples of lossless files include BMP and PNG formats of which PNG will be mainly used for this project.

1.2 AIM

The aim of this project is to develop an LSB image steganography tool using Python. This tool should be capable of both hiding and extracting data from a carrier image. Achieving this involves:

- researching how this technique of steganography is performed in order to recreate it;
- exploring the working principles of already existing LSB steganography tools;
- creating a steganography tool;
- testing the tool by performing steganography.

2 PROCEDURE

2.1 OVERVIEW OF PROCEDURE

The process of creating an LSB steganography tool can be split into multiple parts:

- Setup, which involves installing Python, an IDE, and the essential Python libraries if any of this has not already been done.
- Writing the encoding side, where data is embedded into a carrier image.
- Writing the decoding side, where data is extracted from an image.
- Combining both sides so the result is an easily useable program.

In this project the IDE used was PyCharm. In addition, there are two Python libraries this program required – PIL, which was used for image support, and NumPy which was used for working with arrays when converting data. Rockikz (2021) suggests that the OpenCV library can also be used for image support. After installation, these libraries were imported into the Python file. For details further relating to setup Appendix A.

For the encoding side, the program follows a simple structure. Firstly, the image and data it will work with are read in. The data then has a delimiter that indicates the end of the message attached to it. Both Rupali (2020) and Rockikz (2021) use 5-character delimiters such as '=====' or '#####'; however, Goel (2020) suggests that whether to keep reading the message or not can be stored in every 9th least significant bit. The first 8 bits would contain the binary value of a single character and the 9th bit would contain the value of 1 to keep reading or 0 to stop or vice versa. This method was not used here, as one must know that every 9th bit contains a bit that determines if the message is over or not and not the first bit of the next character in the message which could disrupt the extracting process if a different extraction method other than the same program was used. It also uses more bits to encode data into an image. While delimiters also differ across programs, when extracting the data, finding such a string of characters is still helpful in understanding that the message is over.

Moving on, the program checks if there are enough bytes in the image to encode the given data. Then a copy of the image is made, and the data to encode is converted into a binary format. The program then proceeds to go through each pixel of the image changing the pixel RGB values of the copied image to match the binary digits of the data. It stops this process when there is nothing left to encode. The copied image with embedded data is then saved. Section 2.2. goes more in-depth into the steps of how the encoding side was completed.

For the decoding side, the program reads in the picture it will extract data from and goes through each pixel, reading in their RGB values, and adds 1 or 0 based on the least significant bits of these values to a string that stores binary data. This string is then split into bytes and each byte is then converted into an ASCII format and stored into a string. When it finds the delimiter used for encoding, it stops converting bytes as the full message that needs to be extracted has been found. Finally, the message is displayed. This process is described more in detail in section 2.3.

As the final touch, the main function is defined. It asks for user input which determines whether to encode or decode. Next, it asks for the image it will work with and data to embed into the image if encoding has been chosen. Depending on what the user wants to do, an appropriate function call is made. This considerably improves the usability of the steganography tool.

2.2 PROCEDURE PART 1

The encoding side of the steganography tool was created following the steps described below.

In the main function:

1. Prompts asking the user to provide an input image file name and data to encode were made, the image is then opened using `Image.open()`.
2. Code that attaches a delimiter '***' to the provided date was then written.
3. An equation that calculates the bytes available for storing the data was made by multiplying image dimensions by 3 to get the total number of least significant bits available and then performing floor division by 8 to get bytes.
4. The program was then made to use logic operators to compare the calculated bytes to the length of the string of data to encode as seen in Figure 2-1. If the data strings length exceeds the calculated number of bytes, the program does not perform steganography as there is not enough space in the image.

```
img_bytes = image.width * image.height * 3 // 8
if len(data) > img_bytes:
    raise ValueError("Insufficient bytes. Choose a larger image or provide less data.")
```

Figure 2-1

In the encode function:

5. Utilising the `.copy()` function, code that creates an image copy that will be used to hide data is written.
6. The data that will be hidden is converted to binary using a `to_binary()` function written based on the code provided by Rockikz (2021).
7. A `data_hidden` variable was created and initialised at 0. This is used to keep track of how many bits of data have been hidden and as a string index to see whether 0 or 1 needs to be encoded at any given moment.
8. Image dimensions were retrieved using `.size` function.
9. Nested for loops that run for every pixel in row and column in the image (determined by the retrieved image dimensions) were created. These loops contain the main data hiding code.
10. Based on the current coordinate in the image that is determined by the row and column in the nested for loops `.getpixel()` was used to get the colour values of the pixel at the current coordinate. Steps 9 and 10 can be seen in Figure 2-2.

```
for x in range(0, width):
    for y in range(0, height):
        coordinate = x, y
        pixel = list(enc_img.getpixel(coordinate))
```

Figure 2-2

11. A for loop that runs 3 times (for each colour value – red, green, and blue) was created. Inside of this loop the code checks if data still needs to be hidden – (the `data_hidden` variable is less than the length of the binary data). Pixel values are then modified as necessary.
    An approach that does not involve converting pixel values to binary was used here – an LSB being 1 means that the decimal value of this number is odd, while it being 0 means the number is even. This was used to modify pixels.
    To explain, if a bit that needed to be encoded was 0 and the colour value was odd, the colour value was decreased by 1. If 1 needed to be encoded and the colour value was even it was decreased by 1 (or increased by 1 if the colour value was 0). In other situations, the existing LSB does not need to be replaced as it already matches the message.
    Lastly, the `data_hidden` index was increased.
12. The new modified pixel values are then inserted into the image using `.putpixel()`. The code for steps 11 and 12 can be seen in Figure 2-3.

```
for i in range(0, 3):  # perform actions on red, green and blue pixels
    if data_hidden < len(enc_data):  # check if there is still data to encode
        # change pixel values if needed - even for 0 and odd for 1
        if enc_data[data_hidden] == '0' and pixel[i] % 2 != 0:
            pixel[i] -= 1
        elif enc_data[data_hidden] == '1' and pixel[i] % 2 == 0:
            if pixel[i] != 0:
                pixel[i] -= 1
            else:
                pixel[i] += 1  # makes sure the value doesnt get below 0 if it is originally 0

        data_hidden += 1  # increase hidden data count

enc_img.putpixel(coordinate, tuple(pixel))  # put the modified pixel in coordinate
```

Figure 2-3

13. Steps 10 – 12 are repeated if needed and when the `data_hidden` variable is equal to the binary data length, the loops break.
14. A prompt asking for the name of the new image was made, and code for saving the image was created as seen in Figure 2-4.

```
# save the new image
name = input("Enter the name and extension of the new image: ")
enc_img.save(name, str(name.split(".")[1].upper()))
enc_img.show()  # show how the final image looks
```

Figure 2-4

2.3 PROCEDURE PART 2

The decoding side was created performing actions that are described below.

In the main function:

1. A prompt asking the user to provide an input image file name was made, the image is then opened using `Image.open()`.

In the decode function:

2. Empty strings that will contain extracted binary data and data when it is converted to ASCII values were created.
3. Image dimensions were retrieved using `.size` function.
4. Nested for loops that run for every pixel in row and column in the image (determined by the retrieved image dimensions) were created. As with encoding, these loops contain the main data extraction code.
5. Based on the current coordinate in the image that is determined by the row and column in the nested for loops `.getpixel()` was used to get the colour values of the pixel at the current coordinate.
6. A for loop that runs 3 times (for each colour value – red, green, and blue) was created. It goes through the RGB values of a given pixel and adds 0 to the binary data string if the value is even and 1 if it is odd. Steps 4 – 6 can be seen in Figure 2-5.

```
for x in range(0, width):
    for y in range(0, height):
        coordinate = x, y
        pixel = list(image.getpixel(coordinate))
        for i in range(0, 3):
            if pixel[i] % 2 == 0:
                binary_data += '0'  # if color value is even, add 0 to string
            else:
                binary_data += '1'  # if color value is odd, add 1 to string
```

Figure 2-5

7. Using code suggested by Rockikz (2021), code line for splitting the extracted binary data into bytes and converting each byte to its corresponding ASCII symbol was written. These symbols are stored in the empty data string.
8. When the delimiter '***' is found by the program, the conversion process is stopped as the message has been found. The data is then returned without the delimiter (see Figure 2-6).

```
all_bytes = [binary_data[i: i+8] for i in range(0, len(binary_data), 8)]
for byte in all_bytes:
    data = data + chr(int(byte, 2))
    if data[-3:] == "***":  # look for delimiter to see where message ends
        break
return data[:-3]  # return data without delimiter
```

Figure 2-6

3 RESULTS

To test the results, the finished program was used to perform steganography on images. To demonstrate its functionality, steganography was performed on a PNG image; however, to further test its abilities it was also used on an unfit file - a lossy JPG image. The results are displayed below and further discussed in section 4. For the finished code see Appendix B.

3.1 RESULTS FOR PART 1

This section contains the results of encoding data into different images. Figure 3-1 displays the output produced when running the encoding side of the steganography tool.

```
Select image steganography tool mode:
 1. Encode
 2. Decode
1
Enter PNG image file name: capybara.png
Enter data to encode: Capybaras are short-haired brownish rodents with blunt snouts,
Encoding...
Enter the name and extension of the new image: test1.png
```

Figure 3-1

The above-displayed image displays the program encoding a description into a file provided by the user. Copies of the provided image 'capybara.png' and the modified image can be seen below (Figures 3-2 and 3-3).

Figure 3-2 Original image

Figure 3-3 Modified image

As it can be observed, the pictures cannot be told apart, although one of them contains hidden data in it. The full text that is encoded in the second image is as follows: "Capybaras are short-haired brownish

rodents with blunt snouts, short legs, small ears, and almost no tail. They are shy and associate in groups along the banks of lakes and rivers."

When encoding data into a lossy format, the output following output is produced:

Figure 3-4 Original JPG image

Figure 3-5 Modified image, saved as JPG

Figure 3-6 Modified image, saved as PNG

In this case, Franz Kafka's "As Gregor Samsa awoke one morning from uneasy dreams he found himself transformed in his bed into a gigantic insect." was embedded into the image. It was then saved in both a JPG and PNG format. The result looks identical to the original image, but without performing data extraction it is not possible to tell that steganography on this image was successful. This will be looked at in section 3.2.

3.2 RESULTS FOR PART 2

This section contains the results of extracting data from images that were encoded in section 3.1. Figure 3-7 displays the output produced when running the decoding side of the steganography tool. The program can be seen extracting the same data that was embedded into the image in the previous section which it is successful at.

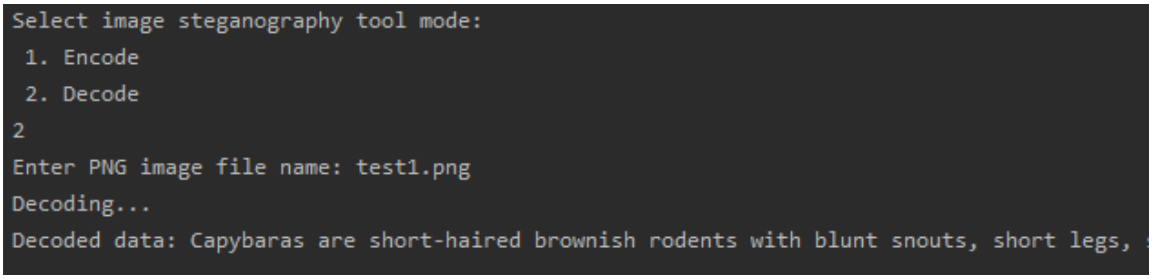

Figure 3-8

When extracting data from the encoded JPG image, it successfully performed the action with the image that was converted to PNG but failed to do so with the JPG image. The output of this can be seen in Figures 3-8 and 3-9.

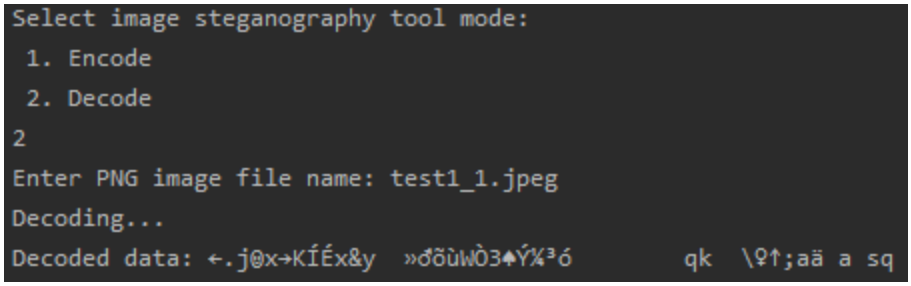

Figure 3-7 Extracted data from a JPG image

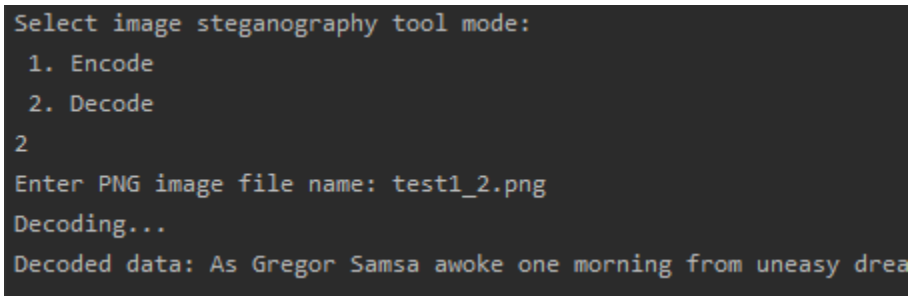

Figure 3-9 Extracted data from a PNG image

As an additional test, an image encoded with another steganography tool was given this program to extract data from; however, as expected, it was unable to do so.

4 DISCUSSION

4.1 GENERAL DISCUSSION

The results of tests performed prove that it is possible to create and utilize an LSB steganography tool independently. On top of that, images the program embedded data in looked the same as the originals, proving just how effective steganography is. The fact that data can be hidden so easily and can be used for malicious intent as mentioned on page 1 closely relates to why being knowledgeable in it is important in cybersecurity. Despite that, it is a quite effective method of security when coupled with other practices such as cryptography as this report discusses.

This project proved successful in developing an LSB steganography tool; however, as seen in the results section it is not comparable to some of the software available for steganography. This tool works best for encoding small amounts of data in PNG images while other available tools can work with larger file sizes and a bigger variety of file types. Some of them also encrypt the payload for extra security. The created tool is not as secure as it encodes data pixel by pixel without scattering the binary data across the image so any outsider could obtain the data by analysing the pixel colours and finding any patterns that could be the encoded data. This tool was also unable to perform steganography on lossy image formats which was not surprising but still a minus. Another disadvantage is that this same tool needs to be used for both encoding and decoding to retain the original message.

4.2 COUNTERMEASURES

The developed tool could benefit from:

- Using bitwise operations on the least significant bits of pixels instead of incrementing or decrementing their values by 1, therefore making the code more efficient.
- Implementing different message end criteria (such as storing whether to keep reading or not in the 9th least significant bit as mentioned on page 4).
- Looking into ways to support different image files.
- Simplifying the usage of this tool, by making the user experience part of the program better as it is very basic.

4.3 FUTURE WORK

Given more time and resources any future work relating to this project could be broadening the tool's usability by implementing other steganography types such as text steganography in it. In addition, data could be encrypted before it is hidden for extra security. Encoding binary digits across the image instead of placing them side by side would increase security even further as the entire message will not be hidden in the same place. Utilizing stego-keys to use this tool in practice and communicate with other parties using steganography is also something worth looking into.

REFERENCES

Semilof, M., Clark, C. (2018) *Steganography*. Available at: https://searchsecurity.techtarget.com/definition/steganography (Accessed 1 May 2021).

Cheddad, A., Condell, J., Curran, K., Mc Kevitt, P. (2010) 'Digital image steganography: Survey and analysis of current methods', *Signal Processing*, 90(3), pp. 727 – 752. doi: 10.1016/j.sigpro.2009.08.010.

Siper, A., Farley R., Lombardo, C. (2005) *The Rise of Steganography*. Available at: http://csis.pace.edu/~ctappert/srd2005/d1.pdf (Accessed 2 May 2021).

Zarelli, N. (2017) *The Wartime Spies Who Used Knitting as an Espionage Tool*. Available at: https://getpocket.com/explore/item/the-wartime-spies-who-used-knitting-as-an-espionage-tool (Accessed 2 May 2021).

Borse, G. Anand, V., Patel, K. (2013) *Steganography: Exploring an ancient art of Hiding Information from Past to the Future.* Available at: https://www.ijeit.com/Vol%203/Issue%204/IJEIT1412201310_33.pdf (Accessed 2 May 2021).

Dickson, B. (2020) *What is steganography? A complete guide to the ancient art of concealing messages*. Available at: https://portswigger.net/daily-swig/what-is-steganography-a-complete-guide-to-the-ancient-art-of-concealing-messages (Accessed 2 May 2021).

Rupali, R. (2020) *Image Steganography using Python*. Available at: https://towardsdatascience.com/hiding-data-in-an-image-image-steganography-using-python-e491b68b1372 (Accessed 3 May 2021).

Shankdhar, P. (2020) *Best tools to perform steganography [updated 2020].* Available at: https://resources.infosecinstitute.com/topic/steganography-and-tools-to-perform-steganography (Accessed 4 May 2021).

Shulmin, A., Krylova, E. (2017) *Steganography in contemporary cyberattacks*. Available at: https://securelist.com/steganography-in-contemporary-cyberattacks/79276 (Accessed 3 May 2021).

Stanger, J. (2020) *The Ancient Practice of Steganography: What Is It, How Is It Used and Why Do Cybersecurity Pros Need to Understand It*. Available at: https://www.comptia.org/blog/what-is-steganography (Accessed 4 May 2021).

Data Recovery Specialists. (2019) *What is the Snatch Ransomware?* Available at: http://www.datarecoveryspecialists.co.uk/blog/what-is-the-snatch-ransomware (Accessed 4 May 2021).

Lu, CS. (2005) *Multimedia security: steganography and digital watermarking techniques for protection of intellectual property.* Hershey: Idea Group Publishing

El-Emam, N. N. (2007). 'Hiding a large amount of data with high security using steganography algorithm.' *Journal of Computer Science*, 3(4), pp. 223–232.

Cheddad, A. (2009). *Steganoflage: A new image steganography algorithm*. Coleraine: University of Ulster

Posey, B. (2015) Redundant. Available at: https://searchstorage.techtarget.com/definition/redundant (Accessed 5 May 2021).

Wu, Z. (2015) *Information Hiding in Speech Signals for Secure Communication* Burlington: Syngress.

Cortes, J. (2019*) Steganography — LSB Introduction with Python — Part 1.* Available at: https://itnext.io/steganography-101-lsb-introduction-with-python-4c4803e08041 (Accessed 6 May 2021)

Scott, A. (2018) *Simple Image Steganography in Python*. Available at: https://hackernoon.com/simple-image-steganography-in-python-18c7b534854f (Accessed 6 May 2021)

Rockikz, A. (2021) *How to Use Steganography to Hide Secret Data in Images in Python.* Available at: https://www.thepythoncode.com/article/hide-secret-data-in-images-using-steganography-python (Accessed 6 May 2021)

Goel, A. (2020) *Image based Steganography using Python.* Available at: https://www.geeksforgeeks.org/image-based-steganography-using-python (Accessed 7 May 2021)

APPENDICES

APPENDIX A

As Python and the PyCharm IDE were already installed on the machine used, the only setup step that was done was installing PIL and NumPy libraries. It was done by following these steps:

1. Going to Settings > Project: [project name] > Project Interpreter
2. Clicking on 'pip' and 'Install Package'
3. Running the commands `pip install Pillow` and `pip install numpy` in PyCharm's terminal.

Steps 1 and 2 only need to be completed if 'pip' has not already been installed.

Setup information for Python and PyCharm can be found in their respective web addresses:

1. https://www.python.org
2. https://www.jetbrains.com/pycharm

APPENDIX B

```
"""
Name: steg.py
Desc: LSB steganography tool
Auth: Ance Strazdina
Date: 05/05/2021
"""

# import libraries
from PIL import Image  # used for image support
import numpy  # used for working with arrays when converting data


# convert data to binary
"""
Code used:
    Title: convert any type of data into binary
    Author: Rockikz, A
    Date: 2021
    Availability: https://www.thepythoncode.com/article/hide-secret-data-in-images-
using-steganography-python
"""


def to_binary(data):
    if isinstance(data, str):
        return ''.join([format(ord(i), '08b') for i in data])
    elif isinstance(data, bytes) or isinstance(data, numpy.ndarray):
        return [format(i, '08b') for i in data]
```

```
    elif isinstance(data, int) or isinstance(data, numpy.uint8):
        return format(data, '08b')
    else:
        raise TypeError("Entered data type is not supported.")


# encode function
def encode(image, data):
    enc_img = image.copy()  # create new image
    enc_data = to_binary(data)  # convert data to binary

    data_hidden = 0  # keeps track of how much data has been hidden, used as string
index
    width, height = enc_img.size

    print("Encoding...")
    for x in range(0, width):
        for y in range(0, height):
            coordinate = x, y
            pixel = list(enc_img.getpixel(coordinate))  # get pixel values in given
coordinate

            for i in range(0, 3):  # perform actions on red, green and blue pixels
                if data_hidden < len(enc_data):  # check if there is still data to
encode
                    # change pixel values if needed - even for 0 and odd for 1
                    if enc_data[data_hidden] == '0' and pixel[i] % 2 != 0:
                        pixel[i] -= 1
                    elif enc_data[data_hidden] == '1' and pixel[i] % 2 == 0:
                        if pixel[i] != 0:
                            pixel[i] -= 1
                        else:
                            pixel[i] += 1  # makes sure the value doesn't get below 0
if it is originally 0

                    data_hidden += 1  # increase hidden data count

            enc_img.putpixel(coordinate, tuple(pixel))  # put the modified pixel in
coordinate

            # break loop if there is no more data to encode
            if data_hidden >= len(enc_data):
                break
        break

    # save the new image
    name = input("Enter the name and extension of the new image: ")
    enc_img.save(name, str(name.split(".")[1].upper()))
    enc_img.show()  # show how the final image looks


# decode function
def decode(image):
    # data strings for extracted and converted data
    binary_data = ''
```

```
    data = ''

    width, height = image.size

    print("Decoding...")
    # go through rgb values in image
    for x in range(0, width):
        for y in range(0, height):
            coordinate = x, y
            pixel = list(image.getpixel(coordinate))
            for i in range(0, 3):
                if pixel[i] % 2 == 0:
                    binary_data += '0'  # if color value is even, add 0 to string
                else:
                    binary_data += '1'  # if color value is odd, add 1 to string

    # split data into bytes and convert bytes to ascii format
    """
    Code used:
        Title: split by 8-bits, convert from bits to characters
        Author: Rockikz, A
        Date: 2021
        Availability: https://www.thepythoncode.com/article/hide-secret-data-in-
images-using-steganography-python
    """

    all_bytes = [binary_data[i: i+8] for i in range(0, len(binary_data), 8)]
    for byte in all_bytes:
        data = data + chr(int(byte, 2))
        if data[-3:] == "***":  # look for delimiter to see where message ends
            break
    return data[:-3]  # return data without delimiter


# main function
def main():
    # choose whether to encode or decode
    mode = int(input("Select image steganography tool mode:\n 1. Encode\n 2.
Decode\n"))

    # encode
    if mode == 1:
        # enter image
        filename = input("Enter PNG image file name: ")
        image = Image.open(filename)

        # enter data
        data = input("Enter data to encode: ")
        if len(data) == 0:
            raise ValueError("No data was entered.")
        data += "***"  # add a delimiter

        # check if there is enough space to perform action
        # multiply image dimensions with 3 to get total bits to encode and use floor
division by 8 to get bytes
```

```
        img_bytes = image.width * image.height * 3 // 8
        if len(data) > img_bytes:
            raise ValueError("Insufficient bytes. Choose a larger image or provide
less data.")

        encode(image, data)  # call the encode function

    # decode
    elif mode == 2:
        # enter image
        filename = input("Enter PNG image file name: ")
        image = Image.open(filename)

        # call decode function and print results
        print("Decoded data: " + decode(image))

    # invalid input
    else:
        raise Exception("Invalid option.")


# call main function
if __name__ == '__main__':
    main()
```