A2 – Stenography

Garik Smith-Manchip

October 7th, 2021

—

COMP 8505

—

Aman Abdulla

D'Arcy Smith

Contents

Introduction

Overview

- Choose language for stenography implementation
- Choose API's to use for implementation
- Design a basic application to hide an image inside an image
- Create a command-line based program
- Two main functions are **Hide** and **Extract**

Goals and Constraints

- Implement for BMP images
- Provide a help function
- Enable **simple** encryption for the image
- Find images large enough to test with
- Structure in three modules; command line processing, image processing and the hiding and extracting of the secret image from the cover image
- Test multiple images
- Show success and failures through data

Design

Language Usage

I chose to use Python 3 for this because I have just started learning this language. I have a longer history using C as my main language, but to provide myself with a more robust learning experience, I will be learning Python along this whole course.

The second reason for this switch is that Python provides simpler libraries to implement the actions needed to complete the assignments in this course. This is good because it allows for focus to be on the concept and understanding stenography and how it works, rather than being confused on how to implement it. The goal is to recreate these programs using C at a further date.

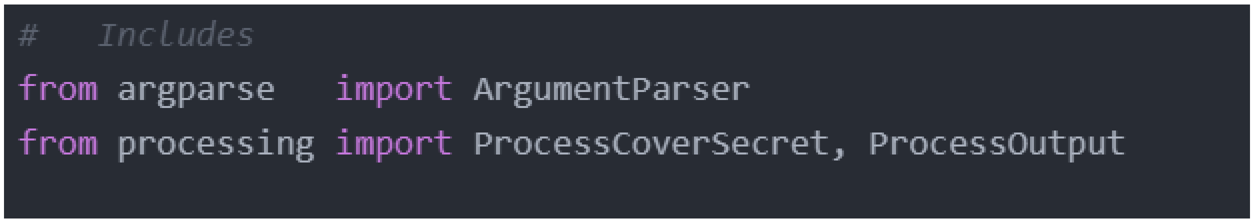
Another reason for the switch is that there is an active library for handling command line arguments gracefully to automatically assign them to variables and give them long and short forms of flags. It creates a help menu as well to assist in usage. This all saves on the physical code I need to write and eases the complicated setup to allow focus on the business logic of the covert channel implementation.

Code Overview

The python code was written after I decided to use the libraries provided though the language.

Main - Main.py

Imports



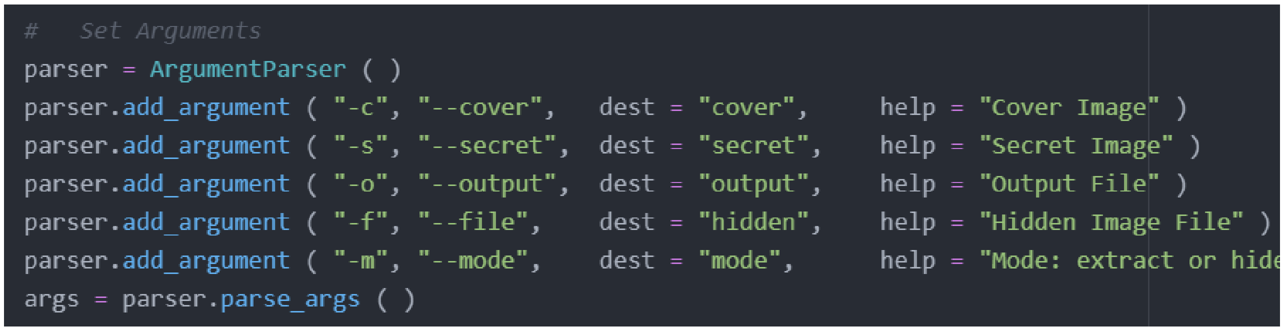
```
#   Includes
from argparse   import ArgumentParser
from processing import ProcessCoverSecret, ProcessOutput
```

The first thing I do is to import an argument parser to handle the inputs from the user. This parser also creates a help menu that has provides a usage report with the **-h** flag.

Next is to import the two functions that the Main program will use to process the input images from the client.

The second reason is a little odder. There is an active library for handling command line arguments gracefully to automatically assign them to variables and give them long and short forms of flags. It creates a help menu as well to assist in usage. This all saves on the physical code I need to write and eases the complicated setup to allow focus on the business logic of the covert channel implementation.

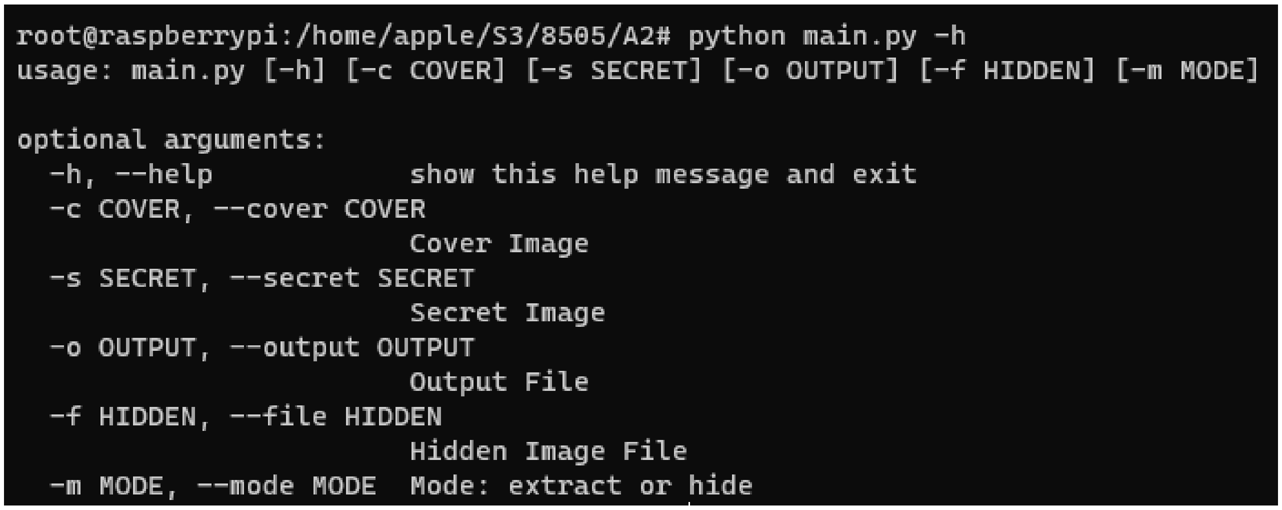
Arguments



```
#   Set Arguments
parser = ArgumentParser ( )
parser.add_argument ( "-c", "--cover",   dest = "cover",    help = "Cover Image" )
parser.add_argument ( "-s", "--secret",  dest = "secret",   help = "Secret Image" )
parser.add_argument ( "-o", "--output",  dest = "output",   help = "Output File" )
parser.add_argument ( "-f", "--file",    dest = "hidden",   help = "Hidden Image File" )
parser.add_argument ( "-m", "--mode",    dest = "mode",     help = "Mode: extract or hide
args = parser.parse_args ( )
```

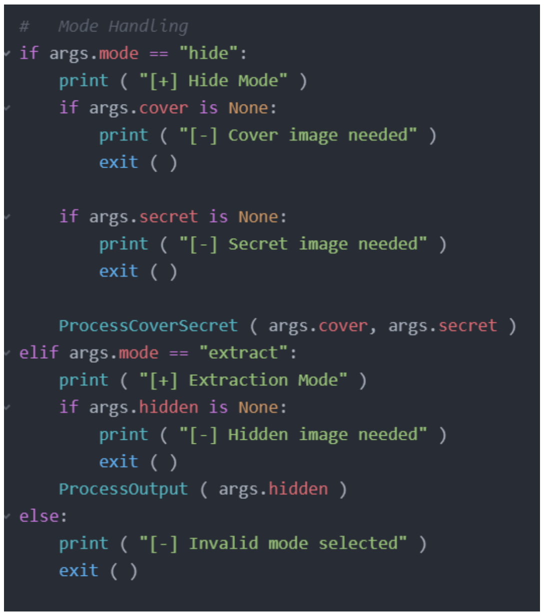
The built-in argument parser served wonders in this assignment. As I am still new to Python, this library allowed me to assign short and long format flags for my arguments, to be listed in any order, while assigned to their own variable to be used later. This implementation also comes with a help menu to help cut down on code that shows usage and error menus.

The users' arguments will be both the cover file and the secret file, inputted as BMP images. The hidden image will then be further processed to be hidden into the Least Significant Bits of the cover image. These are needed if the mode is set to **hide**. If the user wants to uncover a hidden image, they need to provide the hidden image as well as the password for the encrypted image.



```
root@raspberrypi:/home/apple/S3/8505/A2# python main.py -h
usage: main.py [-h] [-c COVER] [-s SECRET] [-o OUTPUT] [-f HIDDEN] [-m MODE]

optional arguments:
  -h, --help            show this help message and exit
  -c COVER, --cover COVER
                        Cover Image
  -s SECRET, --secret SECRET
                        Secret Image
  -o OUTPUT, --output OUTPUT
                        Output File
  -f HIDDEN, --file HIDDEN
                        Hidden Image File
  -m MODE, --mode MODE  Mode: extract or hide
```

Mode Handling

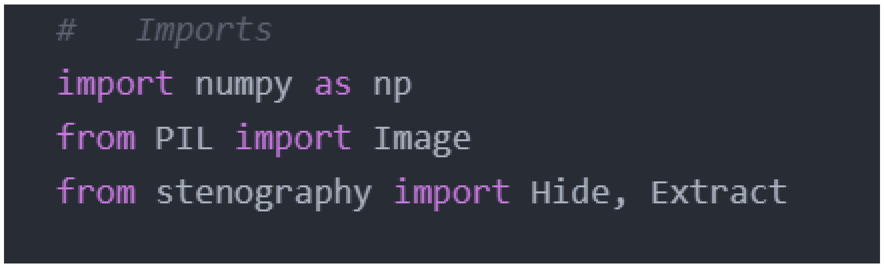


```python
#   Mode Handling
if args.mode == "hide":
    print ( "[+] Hide Mode" )
    if args.cover is None:
        print ( "[-] Cover image needed" )
        exit ( )

    if args.secret is None:
        print ( "[-] Secret image needed" )
        exit ( )

    ProcessCoverSecret ( args.cover, args.secret )
elif args.mode == "extract":
    print ( "[+] Extraction Mode" )
    if args.hidden is None:
        print ( "[-] Hidden image needed" )
        exit ( )
    ProcessOutput ( args.hidden )
else:
    print ( "[-] Invalid mode selected" )
    exit ( )
```

Lastly the Main program handles the type of mode provided and what path to take. The hidden mode needs to be provided with the cover and secret image and the extract mode needs the hidden image.

Image Processing - Processing.py

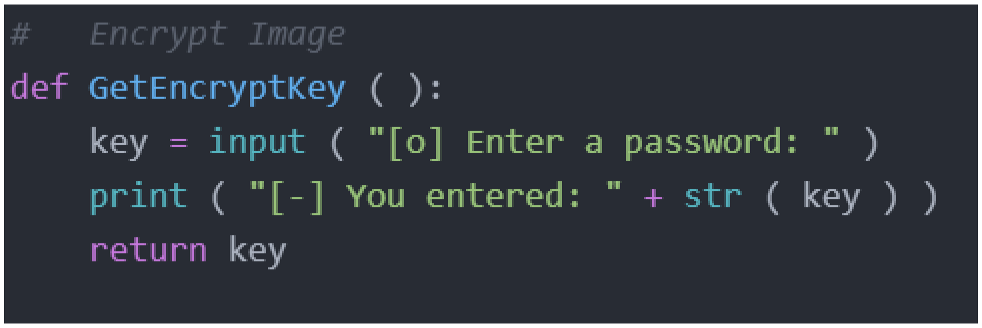
Imports



```python
#   Imports
import numpy as np
from PIL import Image
from stenography import Hide, Extract
```

The imports needed include the Pillow library to process the images as well as the Hide and Extract functions of the actual business logic program. The NumPy library is another image processing import that works with Pillow to process and breakdown images into bits.

Get Encryption Key



```python
#   Encrypt Image
def GetEncryptKey ( ):
    key = input ( "[o] Enter a password: " )
    print ( "[-] You entered: " + str ( key ) )
    return key
```

This simple function gets the user input for the encryption key that needs to be an Integer, and saves it for the actual encryption process. The reason why I only accept integers is because encryption via Python is new to me, and it is a simple process to add to the Stenography image. This implementation saves time for the business logic part of the application.

Image Processing

```
#   Hide Image
def ProcessCoverSecret ( cover, secret ):
    print ( "[+] Processing Images" )

    cover  = Image.open ( cover )
    secret = Image.open ( secret )
    key    = GetEncryptKey ( )

    Hide ( cover, secret, key )

def ProcessOutput ( hidden ):
    print ( "[+] Extracting Image" )

    key    = GetEncryptKey ( )
    Extract ( hidden, key )
```

These two functions allow for the image to be opened and broken down into a bitmap so that the hide and extract functions can work on the least significant bit (LSB) of the images.

Hide and Extract - Stenography.py

Imports

```
#   Includes
import numpy as np
from PIL import Image
```

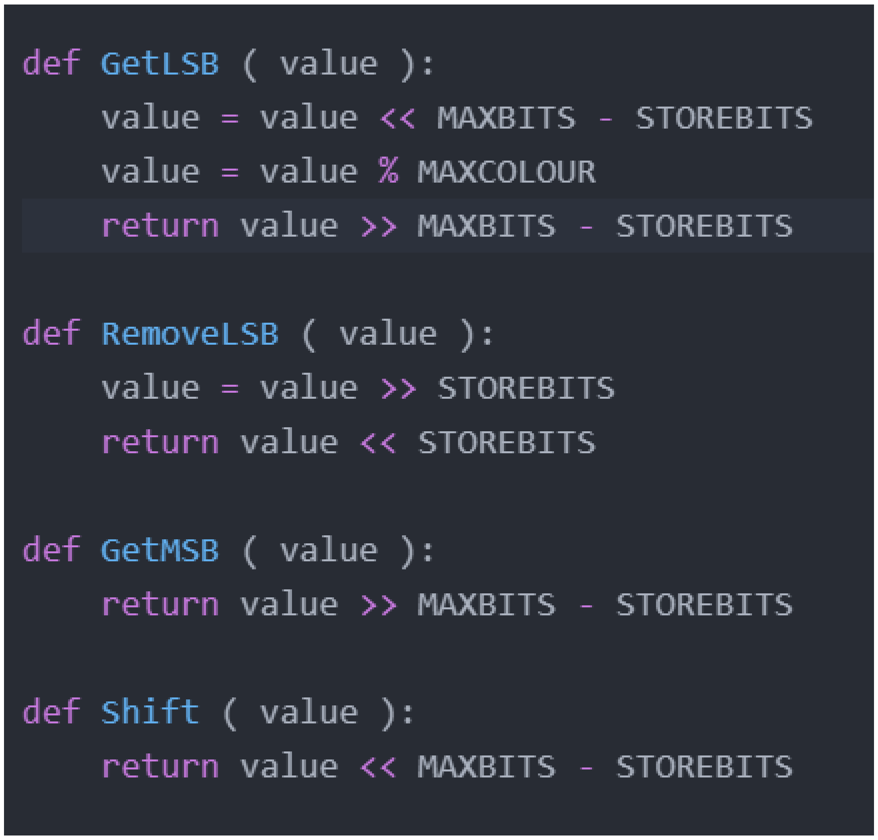
The imports for the image processing stay the same as the image processing program because they need to perform functions on the bits themselves.

Constants

```
#   Constants
MAXBITS    = 8
STOREBITS  = 2
MAXCOLOUR  = 256
```

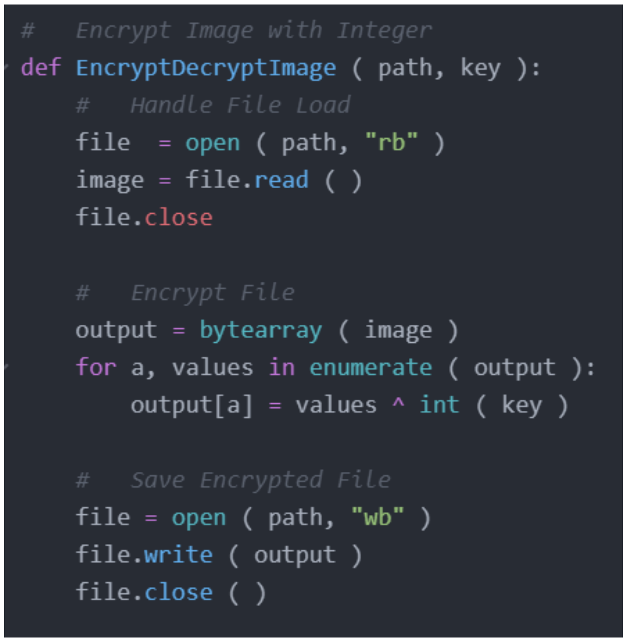
There are three constants used throughout the program. The first being the maximum number of bits in a byte that can be changed. Secondly, the store bits variable will allow the user to edit the number of bits that need to be shifted at the end of the new hidden image to hide the secret image. I chose to set it at the 2 least significant bits because it still hid the image in all three test cases while giving the best quality for the extraction. Lastly, I define the maximum colour bits used for the three channels, red, green, and blue.

Bit Shifting


```
def GetLSB ( value ):
    value = value << MAXBITS - STOREBITS
    value = value % MAXCOLOUR
    return value >> MAXBITS - STOREBITS

def RemoveLSB ( value ):
    value = value >> STOREBITS
    return value << STOREBITS

def GetMSB ( value ):
    return value >> MAXBITS - STOREBITS

def Shift ( value ):
    return value << MAXBITS - STOREBITS
```

These functions handle the number of bits from the cover image and the secret image that need to be moved through the implementation. These bits will be shifted for the three colour channels. The last function does the actual shifting of the bits for the extraction function.

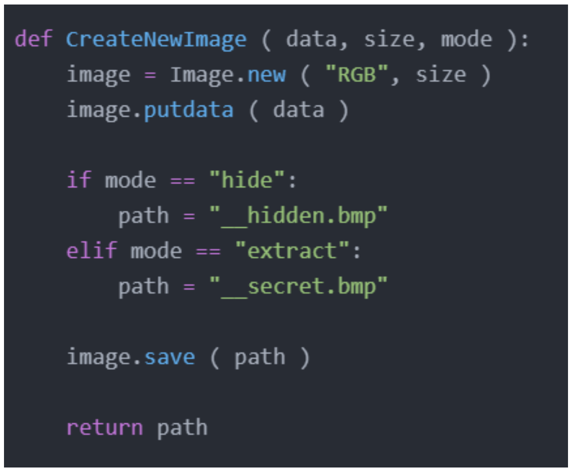
Encryption



```
#   Encrypt Image with Integer
def EncryptDecryptImage ( path, key ):
    #   Handle File Load
    file  = open ( path, "rb" )
    image = file.read ( )
    file.close

    #   Encrypt File
    output = bytearray ( image )
    for a, values in enumerate ( output ):
        output[a] = values ^ int ( key )

    #   Save Encrypted File
    file = open ( path, "wb" )
    file.write ( output )
    file.close ( )
```

This simple encryption function adds the value of the password to the bits of the newly created hidden image. The password can only be an integer because of the exponential nature of the encryption. This function also encrypts the image as well.

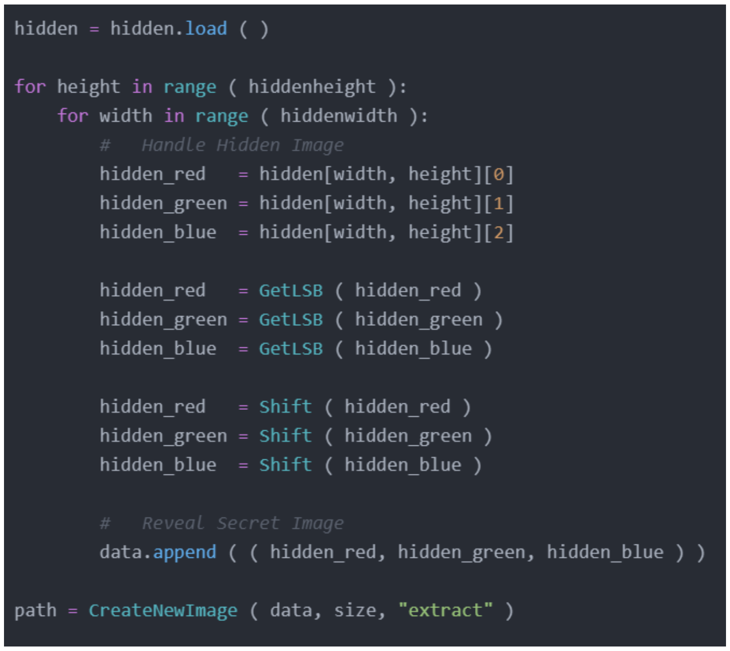
Image Creation



```
def CreateNewImage ( data, size, mode ):
    image = Image.new ( "RGB", size )
    image.putdata ( data )

    if mode == "hide":
        path = "__hidden.bmp"
    elif mode == "extract":
        path = "__secret.bmp"

    image.save ( path )

    return path
```

All this function accomplishes is that it names and exports the new image created by either decrypting and extracting the hidden image to reveal the secret image, or to create a new hidden image from the secret and cover BMP files.

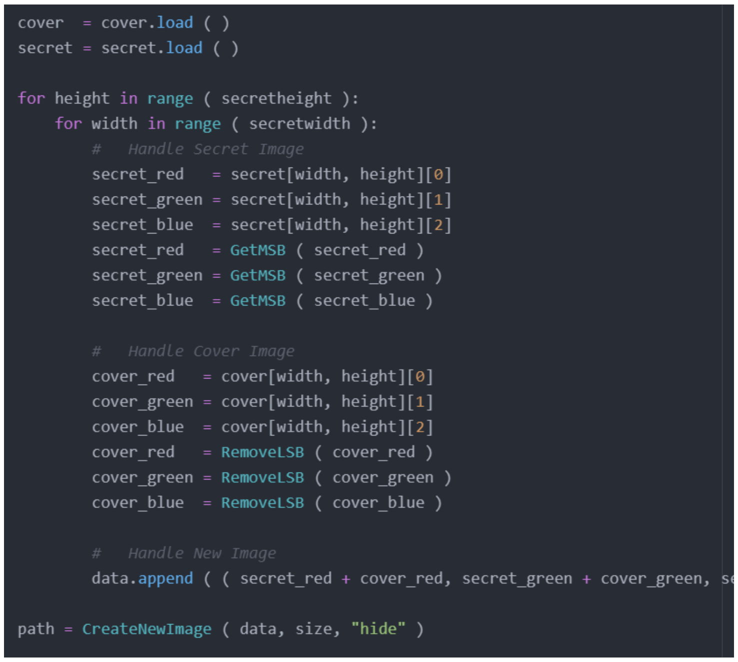
Extract



```
hidden = hidden.load ( )

for height in range ( hiddenheight ):
    for width in range ( hiddenwidth ):
        #   Handle Hidden Image
        hidden_red   = hidden[width, height][0]
        hidden_green = hidden[width, height][1]
        hidden_blue  = hidden[width, height][2]

        hidden_red   = GetLSB ( hidden_red )
        hidden_green = GetLSB ( hidden_green )
        hidden_blue  = GetLSB ( hidden_blue )

        hidden_red   = Shift ( hidden_red )
        hidden_green = Shift ( hidden_green )
        hidden_blue  = Shift ( hidden_blue )

        #   Reveal Secret Image
        data.append ( ( hidden_red, hidden_green, hidden_blue ) )

path = CreateNewImage ( data, size, "extract" )
```

For extraction, the image needs to be filtered through for the changes to the red, green, and blue colour channels and have the bits shifted to create a whole new image. Bits are flipped through the bitmap image where they need to be.

Hide



```
cover  = cover.load ( )
secret = secret.load ( )

for height in range ( secretheight ):
    for width in range ( secretwidth ):
        #   Handle Secret Image
        secret_red   = secret[width, height][0]
        secret_green = secret[width, height][1]
        secret_blue  = secret[width, height][2]
        secret_red   = GetMSB ( secret_red )
        secret_green = GetMSB ( secret_green )
        secret_blue  = GetMSB ( secret_blue )

        #   Handle Cover Image
        cover_red   = cover[width, height][0]
        cover_green = cover[width, height][1]
        cover_blue  = cover[width, height][2]
        cover_red   = RemoveLSB ( cover_red )
        cover_green = RemoveLSB ( cover_green )
        cover_blue  = RemoveLSB ( cover_blue )

        #   Handle New Image
        data.append ( ( secret_red + cover_red, secret_green + cover_green, se

path = CreateNewImage ( data, size, "hide" )
```

The hiding functionality is simple, it loops through the bits of the cover and secret image and creates a new image that looks just like the cover image, with the secret image hidden inside of it. The images area all 32Bit BMP files of all the same size to show what an image will look like in the worst scenario. If an image is hidden inside a larger image, it will be harder to find the difference in pixels.

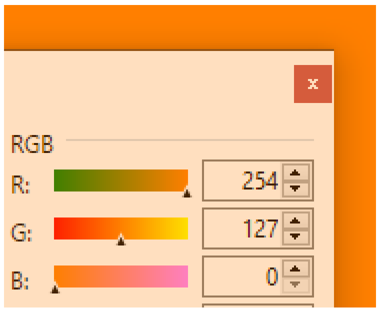
Analysis

Basic Colour Analysis

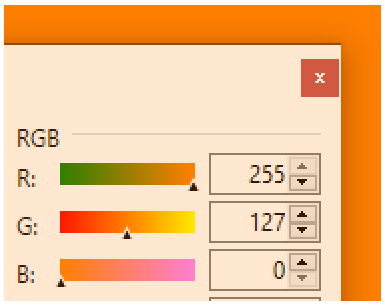
The first image tested was that of the base colour. This is a solid colour that has a picture of a man hidden inside of it. Below are the cover image and the secret image to hide

The stenography program will save the image of the man inside of the orange image.

After the new hidden image is create, it is loaded into Paint. After analyzing the pixels, I notice that there are some that have been slightly changed. Not every pixel is changed but the ones that differentiate between the two images are there.

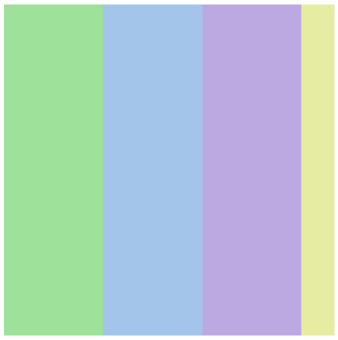
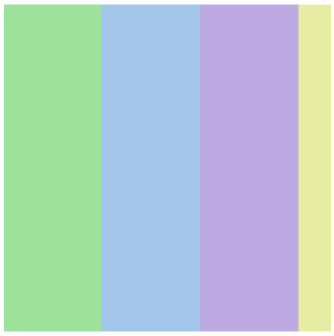


RGB
R: 254
G: 127
B: 0



RGB
R: 255
G: 127
B: 0

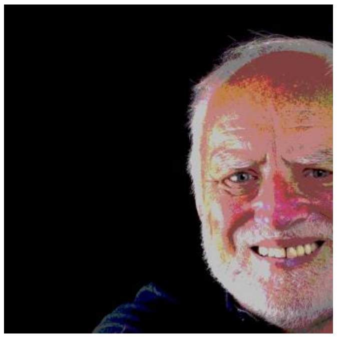
The images are almost exactly alike except for some changes to the colour channels. This was all tested before encryption was included, since the encryption stops the user from being able to load the picture to see it visibly.

These tested cases show a more vibrant array of colours that hide the image even more. These images could allow the program to change its storage bits from 2 to maybe even 4. This is because it will be harder to see the change in images to the human eye.

Image Quality

The main issue with this implementation for the Pillow, NumPy library duo is the image that gets written after decryption is not that of the original.



Side by side, the images are vastly different. This happens because the bits of the colours aren't put back perfectly from when they were taken apart. The quality of this hiding and extraction process pixilates the output image. The output is still close to the original but not of the quality expected from other programs.

This is a testament to the software shown in class. It was able to hide the image and extract it while keeping the expected quality, even if it only slightly diminished.

Test Plan

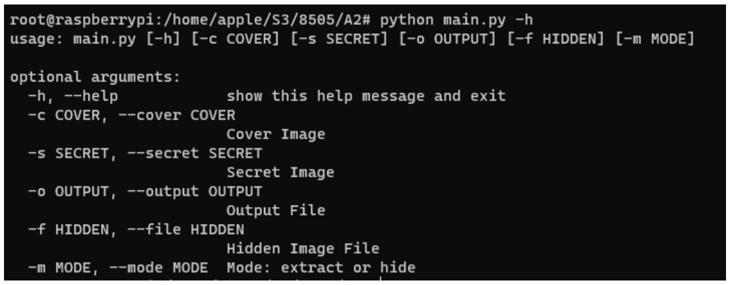
Overview

Case	Machines	Description	Expectation	Outcome
1	Apple	Open help menu	Help menu appears with usage information	Success
2	Apple	Hide image inside cover image	Hides a secret image inside a cover image, encrypting it and sending it to a new file	Success
3	Apple	Extract image from hidden image	Extract the original image from the stenography image after putting in the correct password	Success
4	Apple	Encryption	Test the password encrypted on the image	Success

Test Cases

1 - Open Help Menu

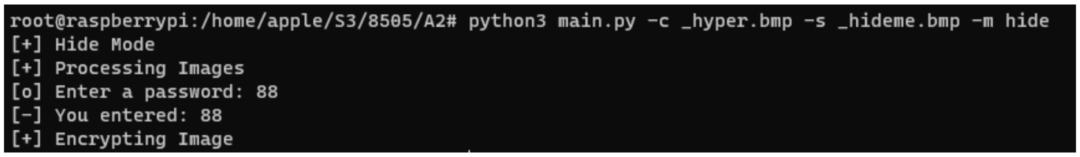
Run `python3 main.py -h`



```
root@raspberrypi:/home/apple/S3/8505/A2# python main.py -h
usage: main.py [-h] [-c COVER] [-s SECRET] [-o OUTPUT] [-f HIDDEN] [-m MODE]

optional arguments:
  -h, --help            show this help message and exit
  -c COVER, --cover COVER
                        Cover Image
  -s SECRET, --secret SECRET
                        Secret Image
  -o OUTPUT, --output OUTPUT
                        Output File
  -f HIDDEN, --file HIDDEN
                        Hidden Image File
  -m MODE, --mode MODE  Mode: extract or hide
```

2 - Hide Image

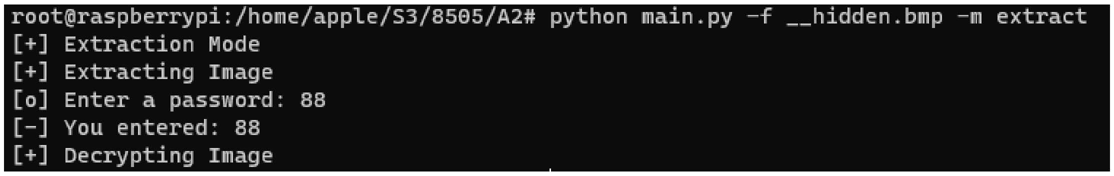
Run python3 main.py -c [cover] -s [secret] -m hide



```
root@raspberrypi:/home/apple/S3/8505/A2# python3 main.py -c _hyper.bmp -s _hideme.bmp -m hide
[+] Hide Mode
[+] Processing Images
[o] Enter a password: 88
[-] You entered: 88
[+] Encrypting Image
```

3 - Extract Image

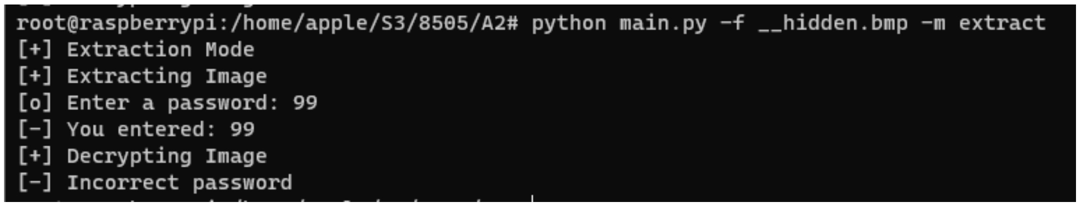
Run python3 main.py -f [hidden] -m extract



```
root@raspberrypi:/home/apple/S3/8505/A2# python main.py -f __hidden.bmp -m extract
[+] Extraction Mode
[+] Extracting Image
[o] Enter a password: 88
[-] You entered: 88
[+] Decrypting Image
```

4 - Encryption - Decryption

Run python3 main.py -f [hidden] -m extract and input an incorrect password



```
root@raspberrypi:/home/apple/S3/8505/A2# python main.py -f __hidden.bmp -m extract
[+] Extraction Mode
[+] Extracting Image
[o] Enter a password: 99
[-] You entered: 99
[+] Decrypting Image
[-] Incorrect password
```

Conclusion

Overview

The process of hiding an image inside an image is unbelievably cool. For a fun exercise, this is exactly what you expect from a hacker, to be able to hide and send images and data discretely so that no one is the wiser.

On a somber note, the general application from an attacker seems to be to transfer data inside an image that might not be legal. The first use case that comes to mind is child pornography. This despicable act can be masked by hiding these images inside basic images and encrypting them so not one can access them except for the criminal.

On a good note, stenography can be used to secretly send data from one organization or person to another. An application of this is sending information about war efforts inside images that an attacker could just assume is a picture of family or friends headed back and forth.

Personal

Personal use stops at the fun of hiding images inside others. It is incredibly fun to show a friend that this is a possibility to even do. Prior to the lecture on stenography, I had no idea this was a possibility.

The more learned become about what an attacker could do, helps my mind stat to process how to properly defend against it. The first thing I think of for illegal activities is to set up a recording device to catch a perpetrator in the act of hiding the image. Another way to limit this is to install monitoring software on the perpetrators machine, using a Rubber Ducky to see if an illicit act takes place.

This could be a fun assignment for a final project, to find a better way to hide these images, possibly looking for similar colours to the processed bit so that it is near undetectable, even loading multiple bits at a time into a cover image.