LSB Steganography Tool

COMP8505 Assignment 2
A00991905
Junyin Xia
Oct 8 2021

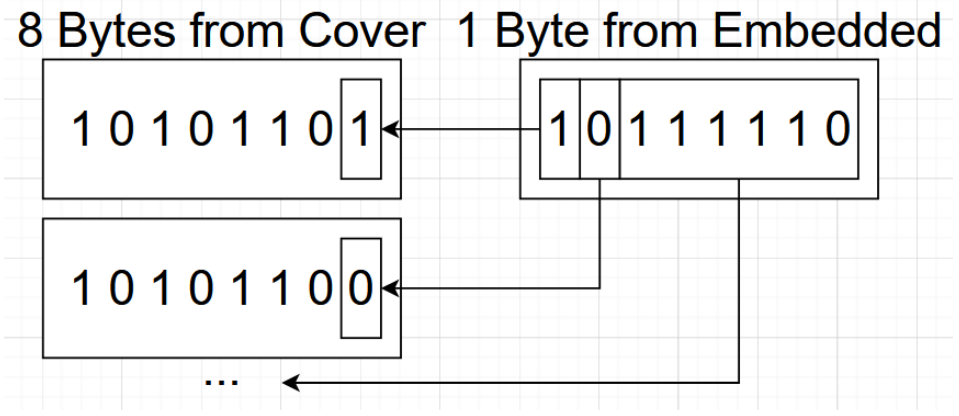
Table of Contents

Introduction

Steganography is the art of hiding an image within another image. For this report, I will be looking at one of the common methods, least bit insertion (LSB). In steganography, there are 3 important concepts:

1. cover medium : (the original image)
2. hidden/embedded data (the image we're trying to hide)
3. stego medium (the output image)

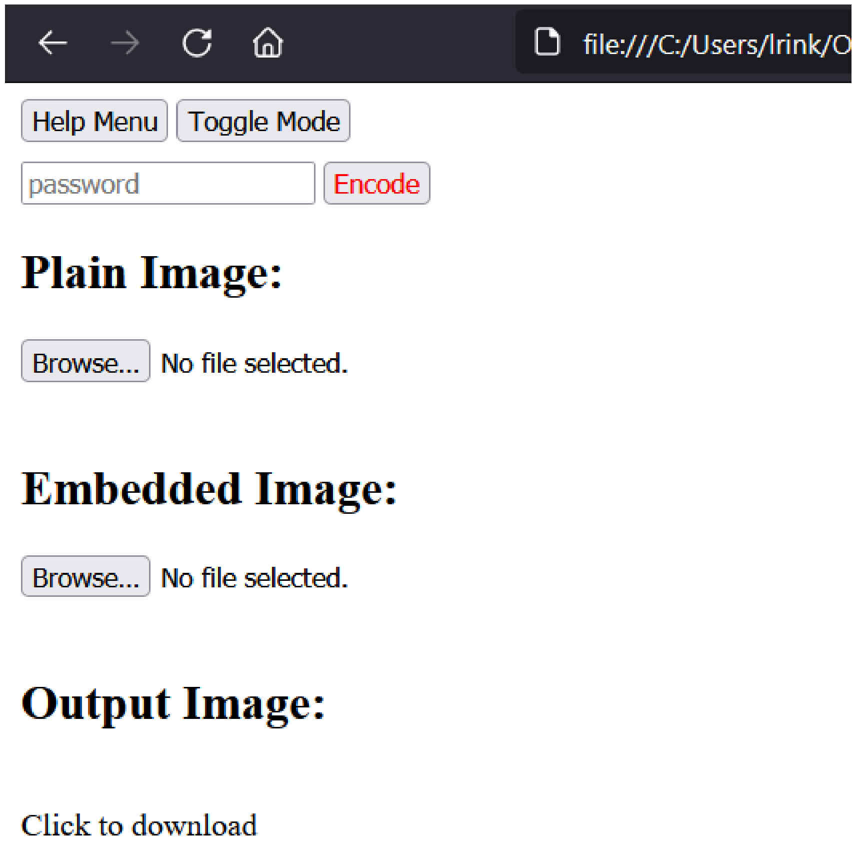
This method hides 1 bit from the embedded as the final bit in 1 byte of the cover medium.



For more technical details and implementation issues of this, see Design Document.pdf

Tool Usage

After opening dcstego.html in a browser, the user will see something like the following



Ui Elements

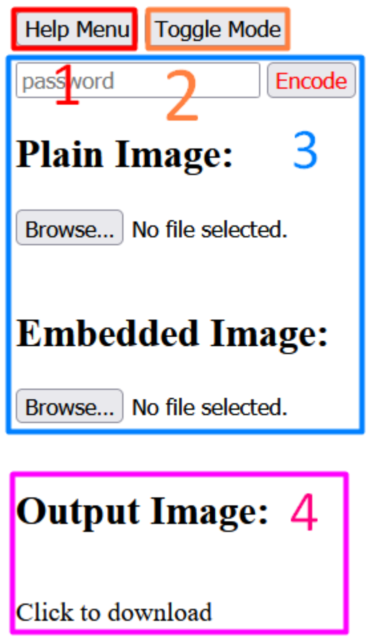


Figure 1: The tool in encode mode. The red button shows "Encode"

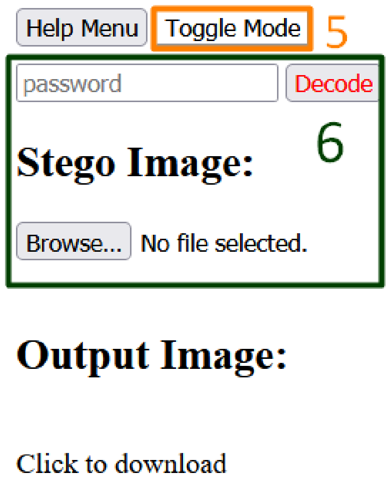
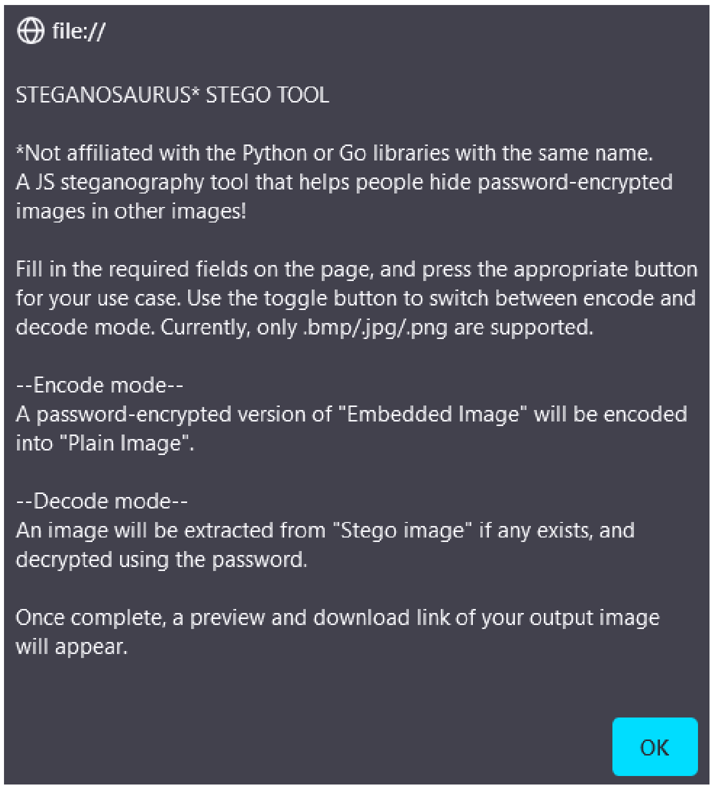


Figure 2: The tool in decode mode. The red button shows "Decode"



file://

STEGANOSAURUS* STEGO TOOL

*Not affiliated with the Python or Go libraries with the same name.
A JS steganography tool that helps people hide password-encrypted images in other images!

Fill in the required fields on the page, and press the appropriate button for your use case. Use the toggle button to switch between encode and decode mode. Currently, only .bmp/.jpg/.png are supported.

--Encode mode--
A password-encrypted version of "Embedded Image" will be encoded into "Plain Image".

--Decode mode--
An image will be extracted from "Stego image" if any exists, and decrypted using the password.

Once complete, a preview and download link of your output image will appear.

Figure 3: The help menu that pops up when "Help Menu" is clicked

1. Help menu. Per requirement a help function was added with usage instructions See Figure 3

2. Toggle mode button, Switches between “Encode” and “Decode” mode. Figure 1 shows the Encode mode UI, Figure 2 shows the Decode mode UI.

3. Encode mode UI elements, enter a password, a carrier image, and your image to imbed. Click encode to automatically encode the image.

4. Output Image, this is present in both modes. The download link will become clickable once an image is processed, either encoded or decoded.

6. The toggle button alternates color between modes. In hindsight I’m not sure why I added this.

5. Decode mode UI elements, enter a password, the encoded stego image,. Click decode to automatically decode the image.

See bmp_demo.mp4 for more.

Tests

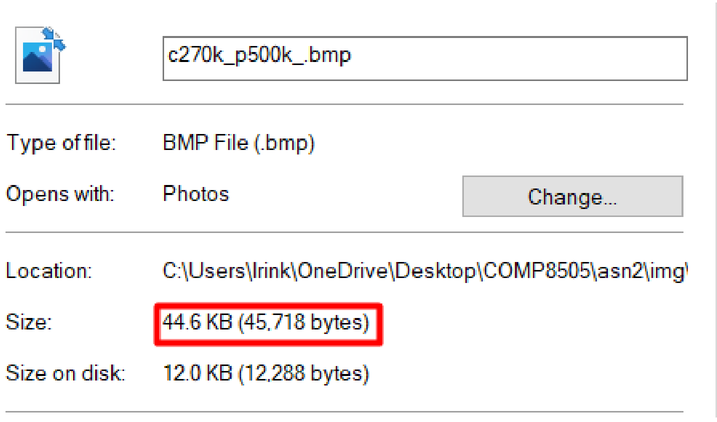
The goal of steganography is to make sure the images don't appear to be obviously modified. Also we need to be able to extract the original image back from the tool. For the demo, I used a short python tool for quickly checking image pixel RBG values. I later rechecked them in paint.net for more accurate values.

```
# pixel_checker.py
from PIL import Image
from sys import argv
if len(argv) != 4 :
    print("Usage:",argv[0],"<image file> <x> <y>")
    exit()
im = Image.open(argv[1])
pix = im.load()
print("WxH:", im.size)
print("RGB:", pix[int(argv[2]),int(argv[3])])
```

#	Goal	Steps	Video	Pass
1	Tool works properly with basic .bmp images	Upload .bmp as cover image Upload .bmp as embedded image Encode and download Run pixel reader on old and new images Upload .bmp as stego image Decode and download Open file in gallery, visually inspect Make sure file size and pixel data are same	bmp_demo.mp4	Yes
2	Tool works with high res images without visual difference, as well as different image formats	Upload high res .png as cover image Upload .jpeg as embedded image Encode and download encoded img Run pixel reader on old and new images Upload encoded .png as stego image Decode and download Open files in gallery, visually inspect Make sure file size and pixel data are same	high_res_demo.mp4	Yes
3	Tool outputs blocky image with solid colour blocks without visual difference	Upload blocky image with solid colurs as cover image Upload solid image as embedded image Encode and download Run pixel reader on old and new images Upload stego image Decode and download Open files in gallery, visually inspect Make sure file size and pixel data are same	solid_demo.mp4	Yes

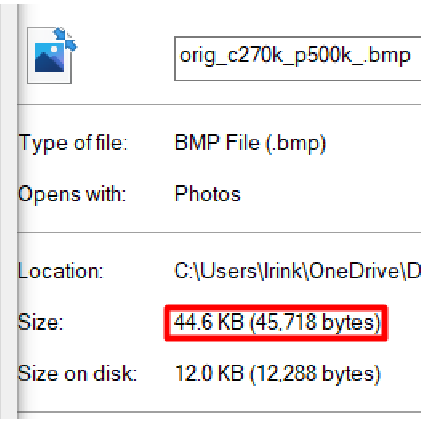
Observations

The file size of the embedded image is preserved when it gets encoded and decoded to a base64 Url.



c270k_p500k_.bmp

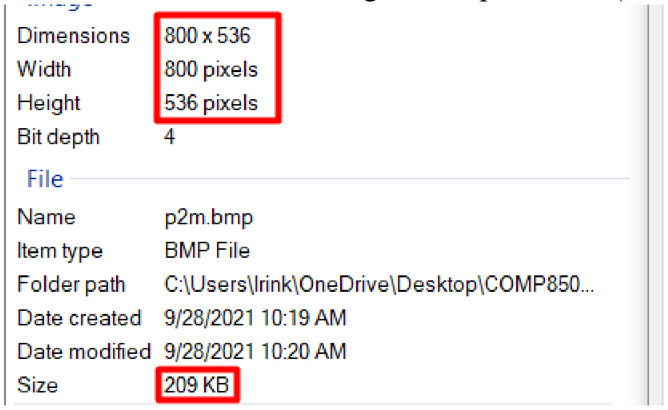
Type of file:	BMP File (.bmp)
Opens with:	Photos
Location:	C:\Users\lrink\OneDrive\Desktop\COMP8505\asn2\img\
Size:	44.6 KB (45,718 bytes)
Size on disk:	12.0 KB (12,288 bytes)



orig_c270k_p500k_.bmp

Type of file:	BMP File (.bmp)
Opens with:	Photos
Location:	C:\Users\lrink\OneDrive\D
Size:	44.6 KB (45,718 bytes)
Size on disk:	12.0 KB (12,288 bytes)

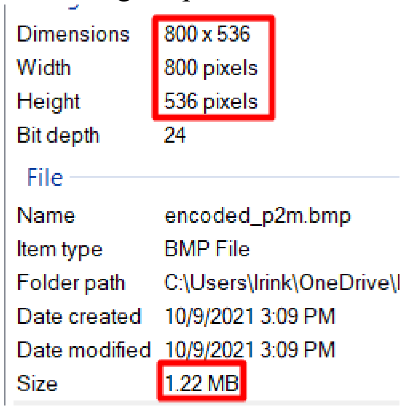
The file size of the cover image is not preserved (the width / height / pixels are however)



Dimensions	800 x 536
Width	800 pixels
Height	536 pixels
Bit depth	4

File

Name	p2m.bmp
Item type	BMP File
Folder path	C:\Users\lrink\OneDrive\Desktop\COMP850...
Date created	9/28/2021 10:19 AM
Date modified	9/28/2021 10:20 AM
Size	209 KB

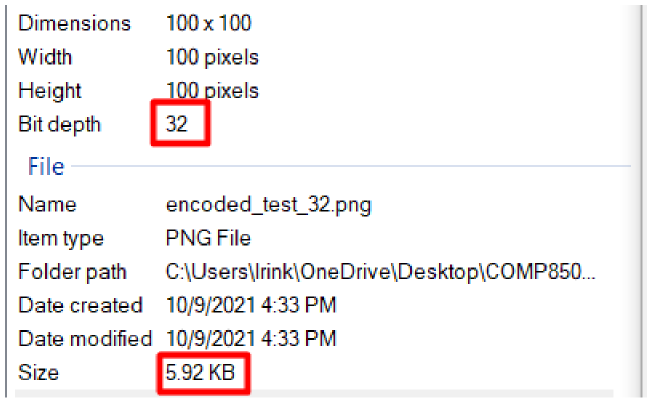


Dimensions	800 x 536
Width	800 pixels
Height	536 pixels
Bit depth	24

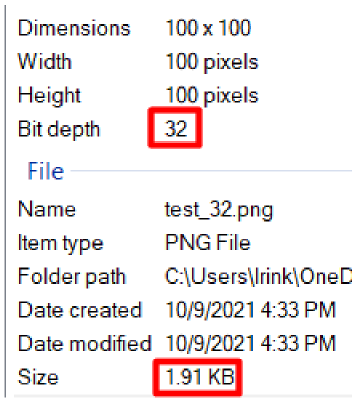
File

Name	encoded_p2m.bmp
Item type	BMP File
Folder path	C:\Users\lrink\OneDrive\
Date created	10/9/2021 3:09 PM
Date modified	10/9/2021 3:09 PM
Size	1.22 MB

At first glance it looks like this is because of the different bit depths (how many bits it takes to represent 1 pixel). The JS canvas class always converts to 32bit images (8bits per channel, RGBA has 4 channels). I ran a test using 32bit images, and the bit depths turns out to not be a factor:



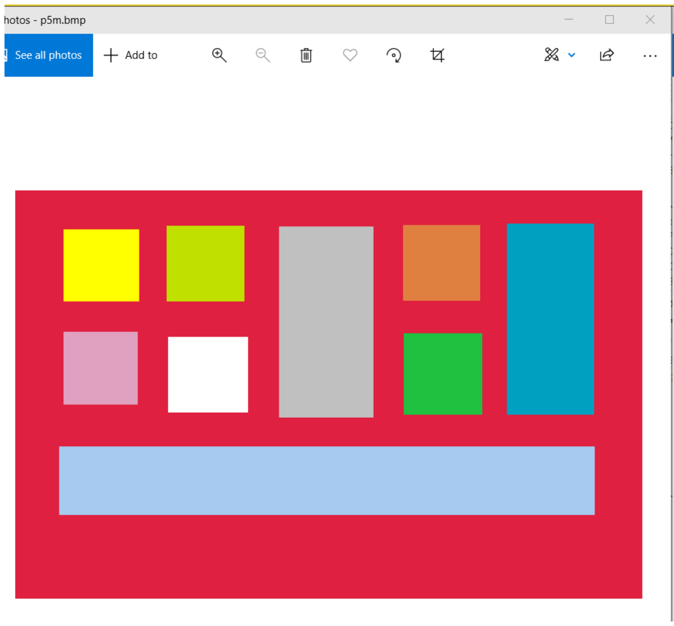
Dimensions	100 x 100
Width	100 pixels
Height	100 pixels
Bit depth	32
File	
Name	encoded_test_32.png
Item type	PNG File
Folder path	C:\Users\lrink\OneDrive\Desktop\COMP850...
Date created	10/9/2021 4:33 PM
Date modified	10/9/2021 4:33 PM
Size	5.92 KB

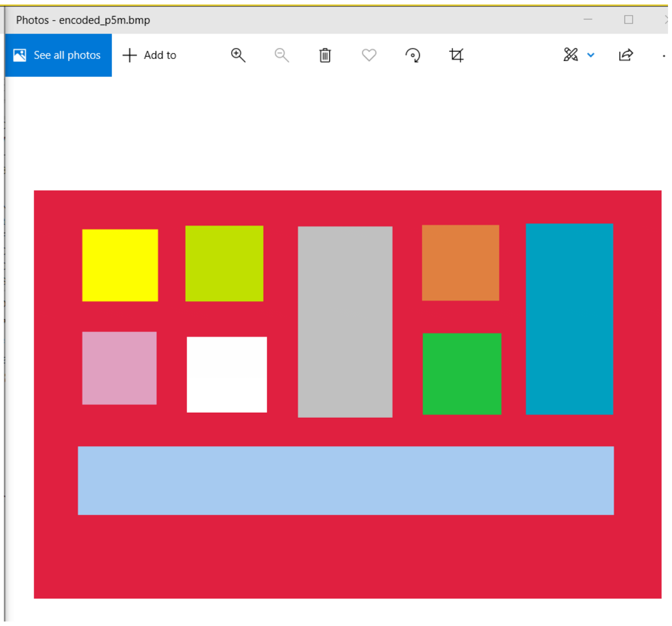


Dimensions	100 x 100
Width	100 pixels
Height	100 pixels
Bit depth	32
File	
Name	test_32.png
Item type	PNG File
Folder path	C:\Users\lrink\OneD
Date created	10/9/2021 4:33 PM
Date modified	10/9/2021 4:33 PM
Size	1.91 KB

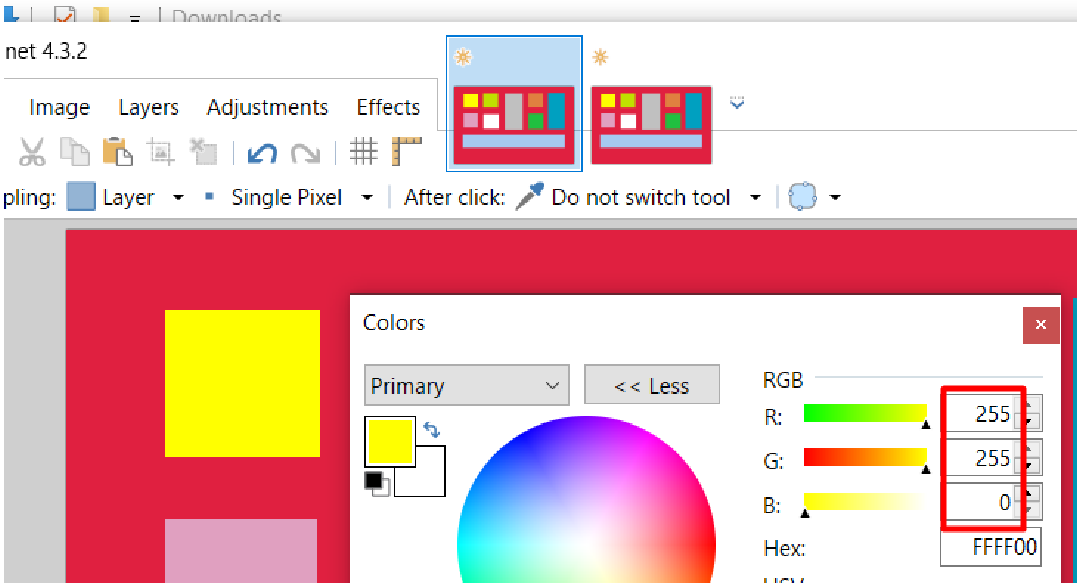
As such, I suspect there's some browser optimization done on the original image (related to the alpha channel issue mentioned in my Design Document.pdf) where the browser creates essentially a new image using the same pixel values and mime type as the original, but uses its own encoding everything else in the file, which results in different file size. The same thing happens when you pass an image through a jpg to png converter. Despite this drawback, JS is still good enough to use for a LSB steganography channel as shown in the tests.

Visually, the original and the output images are identical, regardless of if bright colours are used.



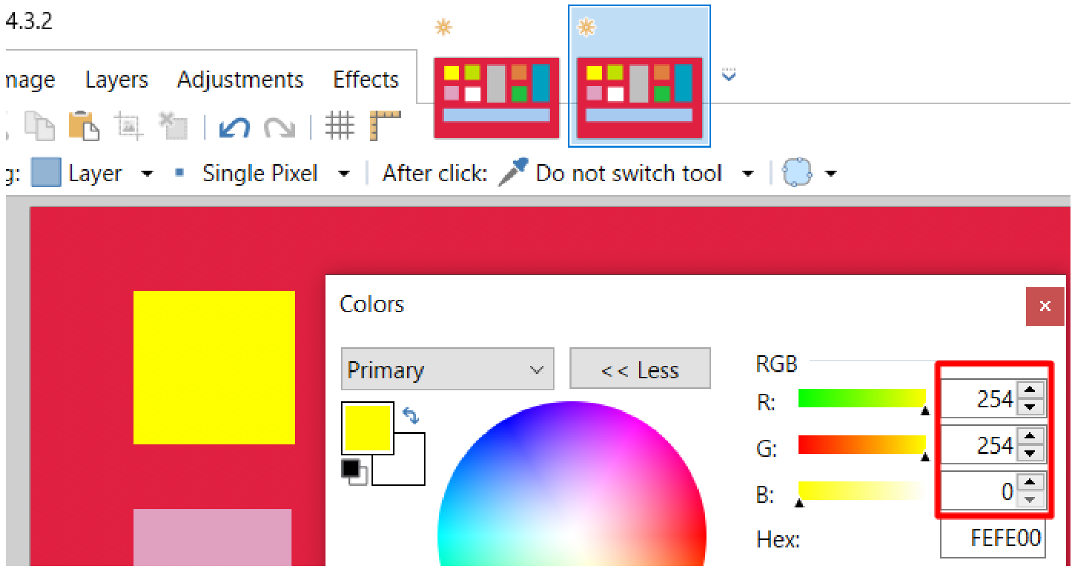


It's only when you look at the pixel data that there's a difference (the difference in value comes from the browser optimization)



Sampling: Layer
Single Pixel
After click: Do not switch tool

R: 255
G: 255
B: 0
Hex: FFFF00



Sampling: Layer
Single Pixel
After click: Do not switch tool

R: 254
G: 254
B: 0
Hex: FEFE00

Conclusion

I now have a better understanding of how images and urls work. I now have experience with bitwise operators, and moving data between bytes. Although the LSB insertion method is flawed (only 1bit per byte could be used for payload, the colours are numerically off, etc), it is impossible to identify them even with a trained eye. An analyst would have to be tipped off about an image to identify it as a carrier. Even then, if it’s password encrypted, it would still be extremely unlikely for them to be able to retrieve the embedded image.