Owen Shelton

CSCI 8810 Project 3

Spring 2011

Contents

[Introduction 2](#_Toc289848141)

[Experiments 4](#_Toc289848142)

[Conclusions 7](#_Toc289848143)

# Introduction

I found this to be an interesting project. I have no experience with cryptography or security, but the idea of hiding data in images appeals to me greatly and I see it as a kind of hiding in plain sight technique. Implementing the least significant bit technique was also surprisingly simple. For the purposes of my experiments I used Figures 1 and 2 to hide data in.

|  |  |
| --- | --- |
| Figure 1, 512x385 | Figure 2, 768x576 |

Figure 1 should be recognizable from previous projects. Figure 2 is a picture I took about a month ago at the State Botanical Gardens. Figure 1 contains large areas of relatively uniform color while Figure 2 has lots of variation and very little uniformity. For my experiments with text I actually used the Chrisitan Bible. It is a very large book and it is very easy to determine how much text is encoded. Figures 3 and 4 show two of the images I embedded. Both are technically binary images, but due to the way wxWidgets handles image data they are automatically expanded into RGB color space.

|  |  |
| --- | --- |
| Figure 3, 512x512 | Figure 4, 64x64 |

Figures 3 and 4 are both images that I used in Project 2 to help me verify the correctness of my implementation of Connected Component Labeling. The fact that they are binary helps to make any errors in extraction stand out as any mistake in extraction would likely result in a pixel being something other than black or white. Unfortunately these images are also a kind of worst case for my implementation as large uniform areas of black and white pixels will result in a kind of banding effect when embedded into an image. This is due to the fact that white and black pixels affect each pixel in the cover image in the exact same way creating a noticeable shift in intensity over uniformish areas. My application does not automatically detect when a grayscale image is having data embedded into it and will therefore not take extra steps to ensure a grayscale image remains grayscale. I also pass the dimensions of the embedded image and the length of the text along with the embedded data rather than embedding it in the image itself. The extraction and embedding procedure for both text and images is both virtually identical as a string is nothing more than an array of characters (bytes) and wxWidgets provides a method to extract the pixel data from an image as one long array of unsigned characters.

# Experiments

Before I explain some of my results it is important to know how to determine how much data can be stored in an image. This is quite easy to calculate and the formula is:

Where *Nmax* is equal to the maximum number of bits that can be stored and *Nused* is the number of least significant bits in the destination image to be used. *Width* and *Height* are the dimensions of the destination image and the multiplication by 3 accounts for the fact that all images in wxWidgets have three channels (RGB) regardless of the format of the image format and also apply to images that are being embedded. Table 1 shows just a few examples to illustrate the formula.

|  |  |  |  |
| --- | --- | --- | --- |
| Image Dimensions | Number of Least Signifigant Bits Used | Number Bytes | Number Pixels |
| 128x128 | 1 | 6,144 | 2,048 |
| 128x128 | 2 | 12,288 | 4,096 |
| 512x512 | 1 | 98,304 | 32,768 |
| 512x512 | 2 | 196,608 | 65,536 |

Table 1

Based on Table1 it can be seen that a tremendous amount of text information can be stored in an image (1 byte typically equals 1 character for ASCII encoded text). Image data does require significantly more space however and that the amount of space required to store a 128x128 RGB image would require 49,152 bytes (or 16,384 pixels).

As the visual results of embedding either text or image are the same, I see little reason to distinguish between them and will instead discuss my results in general. I have found that what images make good images to use is a very difficult question. If the intention is to hide data from the human eye, then even an image such as Figure 1 will work provided that no more than one least significant bit is used. Making a cover image pass through a stego detector is an interesting problem as that would require having access to the programs themselves or having knowledge of what techniques they use, but it would likely be a safe assumption that any steganalysis program would be much more sensitive to color irregularities than the human eye. Figures 5 and 6 show the results of embedding Figure 3 into Figure 1 with both 1 and 2 least significant bits used and Figure 7 is the same area of the original image. All are cropped to show only the top left corner of the image. In Figure 5 the image simply looks a little noisier than the original, but in Figure 6 the regular bands in the left and middle part of the image clearly show that something is going on. Figure 8 shows the same thing as Figure 6, but with 15,000 noisy pixels added before the image is embedded. In this case the noise does distract the eye, but the banding effect is still discernable. Using Gaussian noise may provide better results.

|  |  |
| --- | --- |
| Figure , Figure 3 embedded with 1 least significant bit used. | Figure , Figure 3 embedded with 2 least significant bit used. |
| Figure , Corner of Figure 1 | Figure , Figure 3 embedded with 2 least significant bit used and noise added before embedding. |

As can be seen in these results using images with large uniform areas can make it very easy for the human eye to detect that it could be a cover image, but that when only one least significant bit is used the human eye can be easily deceived and will simply ignore any inconsistences.

Figures 9 and 10 show similar tests using Figure 2 as the image data will be embedded into. For these experiments 3 and 4 least significant bits were used.

|  |  |
| --- | --- |
| Figure , Figure 3 embedded with 3 least significant bits used. | Figure , Figure 3 embedded with 4 least significant bits used. |

I did not include a picture of Figure 2 with less than 3 least significant bits used as there was no appreciable visual difference when compared to the original image. Both figures have been cropped to focus on the walkway which is the area most likely to display artifacts that would give it away as being a cover image as it is the only uniformish area in the image. Figure 9 shows some minor banding on the pathway but no appreciable artifacts anywhere else. Figure 10 is a different story as the banding is clearly seen on the path as well as in the shadowy areas and on the tree bark of the larger tree.

Embedding text was an interesting exercise. Steganography truly shows its usefulness when being used to embed text. A lot of textual information can be conveyed in a relatively small image. For example in the case of Figure 9, I was able to embed the Bible from Genesis 1:1 through Leviticus 25:19, which is roughly 103 pages (exact page numbers vary from Bible to Bible based on publisher and format). In the case of Figure 4 I was able to embed through Numbers 29:23 which is approximately 137 pages. That is a lot of potentially sensitive information.

# Conclusions

I was never able to get satisfactory results from any image out of my considerable collection when using more than 4 least significant bits. I also did not try any synthetic images such as CG renders or the like, but I feel that since these images typically lack the natural noise that even the most uniform photographs have that they would likely perform very poorly.

I think that one way to improve the security of the simple technique would be to spread the data being embedded over the image in a semi-random fashion using some form of 2 dimensional hashing. Obviously if the data being embedded is close to reaching the maximum capacity of the cover image then this method will likely not help much. Also applying compression and encryption to the data before embedding it would certainly be a good idea and make it either more difficult to detect or if detected less likely to be compromised if extracted.

I looked into some programs for detecting and extracting data from a cover image and found very little valuable information. The tools that discussed the techniques they used typically relied on having the program that performed the embedding on hand to make before and after comparisons of the cover image, and only performed detection (<http://www.outguess.org/detection.php>). Some statistical analysis techniques can also use that rely on statistics and color palettes, but noisy images would likely result in false positives (<http://www.uri.edu/personal2/imarcus/stegdetect.htm>).