

# **A DEMONSTRATION OF DIGITAL STEGANOGRAPHY**

By

Cheri B. Walker-Owens

An Honors Project submitted to the University of Indianapolis Strain Honors College in  
partial fulfillment of the requirements for a Baccalaureate degree “with distinction.”

Written under the direction of Dr. Paul Talaga.

March 12, 2018

*Approved by:*

---

*Dr. Paul Talaga, Faculty Advisor*

---

*Dr. Jennifer Camden, Interim Co-Director, Strain Honors College*

---

*First Reader*

---

*Second Reader*

## Abstract

Best practices in steganography recommend that algorithms used to embed a file within an image cause as few visible alterations to the image as possible. The algorithm developed in this research project uses LSB insertion to alter the least significant bit of each byte in the image and replace them with the bits containing the file to be embedded within the image. The focus of the project was to create a program successfully demonstrating best practices using LSB insertion steganography. A survey was administered after the completion of the program to verify that there was no visible difference between images before and after they had files embedded within them. Results suggest that there was no visible difference for color images.

## List of Tables

Table 1: Bitwise operator functions in C++	9
Table 2: Survey responses	16

### List of Figures

Figure 1: Comparison of an image with an embedded photo to its original	3
Figure 2: Visual illustration of the concept of LSB insertion	8
Figure 3: Visual illustration of LSB insertion being implemented in C++	11
Figure 4: Before (left) and after (right) inserting a text file	12
Figure 5: Before (left) and after (right) inserting an executable file	13
Figure 6: Before (left) and after (right) inserting a picture (middle)	13
Figure 7: Usage menu	15
Figure 8: Help menu	15

## Table of Contents

Cover Page	i
Abstract	ii
List of Tables	iii
List of Figures	iv
Table of Contents	v
Statement of Purpose	1
Introduction	2
Method/Procedure	7
Product Produced	11
Results	15
Analysis/Conclusion	18
Reflection	19
References	22
Appendices	23
Appendix A: IRB Exemption Letter	23
Appendix B: Project Source Code	24
Appendix C: Survey	28

### Statement of Purpose

Steganography is defined as the process of hiding a secret message through seemingly inconspicuous means. The concept originated in Ancient Greece with the first reported use of steganography being by Greek historian, Herodotus, involving written messages on the wood bases of wax tables which would then be covered with wax, appearing blank (Kellen, 2001, p. 1). Other methods have included invisible ink, using objects such as beads or dice, and even hiding messages in hard boiled eggs (Zielińska et al., 2014, p. 90). Presently, steganography refers to the process of hiding information within seemingly inconspicuous files such as images or audio recordings (more specifically called digital steganography). The process involves altering a file's bits to hold the message or secretly appending the message to the file in a way that it is not noticeable to users unaware of the message. Altering bits of an image file can make small, imperceptible changes to the picture or large, visible changes depending on which bits (any how many) are changed. The best practice of steganography is to only change enough bits that the difference is not noticeable to the human eye (Villinger, 2011). The purpose of this project was to demonstrate how digital steganography works by writing software that embeds and un-embeds files from image files.

Digital steganography is a newly developing field of study and the practice can be hard to detect. It can be used for malicious means, such as to illegally transmit sensitive data, circulate malicious software, or to coordinate terrorist attacks. Steganography is believed to have possibly been used in the 9/11 attacks in 2001; communications could

have passed unnoticed for as long as three years (Zielińska et al., 2014, p. 88). There are not many ways to combat malicious use of steganography because of the lack of knowledge about it and due to it being “hard to ‘crack’” (Villinger, 2011). At the time of starting this project, most of the research on steganography was classified.

It is important to know more about steganography in order to combat the potential threats posed by it, as is the case with most cyber security threats. An adequate cyber security professional must know how criminal activities are carried out in order to create effective defenses against them. The knowledge and skills gained from this project has served as a valuable experience in not only how steganography works, but in how computers handle and store information. The critical thinking and problem skills required to create this project are also a valuable addition to the pursuit of a career in information security.

## Introduction

Any method of secret communication can be considered steganography (Murphy, 2004, p. 1). The goal of digital steganography is aimed at tricking the human eye into believing that the cover file has not been manipulated (Zielińska et al., 2014, p. 92). Steganography itself is not malicious, but can be used for malicious intent (Murphy, 2004, p. 1). S.D. Murphy explains the basics of simple steganography:

All digital files are made up of bits, which are just ones and zeros. A grouping of eight bits makes up a byte (Example of a byte: 0-1-0-1-0-1-0-1). The most common process of embedding files is based upon the idea that the last bit in each byte adds such a small amount of identity to the

overall file that it could be modified without causing much visual or auditory change to the original file. New information could be stored in this last bit position of each byte until enough storage space is available to store a stolen classified document or a digital photograph taken by a spy. Considering that a Powerpoint file could easily be 10 megabytes in size, if the last bit of every byte was deleted to free up memory space for electronic bits of a hidden file, there would be 1.25 megabytes available (1/8th of the original file size) to hide data. This much space could store several Microsoft Word documents, multiple digital photographs, or even a short video clip. (2004, p. 2)

Murphy also shows a demonstration of what successful steganography looks like.

He explains, “embedding the Pentagon photograph was accomplished using freely downloadable Steganography tools (Steghide, by Stefan Hetzl) on a home computer in just a few minutes. Notice undetectable changes to final embedded photo (carrier file)” (Murphy, 2004, p. 3).



Figure 1: Comparison of an image with an embedded photo to its original (Murphy, 2004, p. 3)

Steganography can be used for both lawful and unlawful purposes. Some legitimate uses include computer forensics, copyright protection (watermarking), and to



bypass censorship and surveillance; illegitimate uses include industrial espionage, cyber weapon exchange, criminal communication and the exposure of sensitive data (Zielińska et al., 2014, p. 88). Steganography users can also encrypt messages before using steganography tools to embed them to add extra security (Wang & Wang, 2004, p. 78). More recently, steganography users have been found to hide viruses and malware as well. This is known as Stegware, defined as “the weaponization of steganography by cyber attackers” (Wiseman, 2018). Since the malware is embedded in files that look safe, it is easy to bypass anti-malware programs. IBM X-Force reported a six fold increase in steganography-based attacks in September of 2017 (McMillen, 2017). A common attack involved embedding the malicious code in an image file, which would activate upon the file being opened by the user (McMillen, 2017). One such attack used an image file to execute code on the victim’s computer that would mine cryptocurrency for the attacker (McMillen, 2017).

With the emergence of steganography, the need to be able to detect hidden information has become a more relevant issue. The detection of steganography is referred to as steganalysis (Zielińska et al., 2014, p. 88). Steganography tools leave traces in the resulting file, making detection possible (Wang & Wang, 2004, p. 80). There are two approaches to steganography detection: visual analysis and statistical (algorithmic) analysis (Wang & Wang, 2004, p. 80). The goal of visual analysis is to reveal the presence of steganography using the naked eye or the assistance of a computer (Wang & Wang, 2004, p. 80). This is done by comparing the original image to the image with the hidden message, or “stego image” (Conway, 2003, p. 55). Statistical analysis is more

powerful and reveals alterations not visible to the human eye (Wang & Wang, 2004, p. 80). Effective steganography detection software is difficult to achieve. Since there are many different ways to embed a message into a file, detection software needs to be versatile and able to detect a wide variety of steganographic methods within a reasonable time frame (Wang & Wang, 2004, p. 81). New methods are constantly being developed and since detection software is developed based off of existing approaches, “any system considered secure today may be broken tomorrow using new techniques” (Wang & Wang, 2004, p. 81).

Embedding files within files is a relatively easy concept to grasp and there are several free steganography tools online. Anyone can download these and use them. That, combined with the difficulty in detecting steganography causes concern for national security. Specifically, it is believed that steganography may become a tool for terrorism, if it is not already. After the events of 9/11, it was believed that the terrorists involved used steganography to communicate with each other. Louis Freeh, former FBI director, expressed in a Senate panel his belief that Osama Bin Laden and followers hid maps, pictures and plans using steganography (Murphy, 2004, p. 4). The FBI found rented hotel rooms used by hijackers that provided 24-hour internet access (Murphy, 2004, p. 4). There is no direct evidence linking the attacks to the use of steganography, however, it is still possible according to Tom Kellen (2001, p. 2). Image files could have been used to hide maps and photographs of targets which would then have been posted onto public websites such as eBay or Amazon (Kellen, 2001, p. 2-3). There are multiple ways to create free websites, which could be used to post steganographic files. These fake

websites would be hard to find without knowing of their existence beforehand (Kellen, 2001, p. 3). Legitimate websites that sympathize with terrorists or have employees who sympathize can also be used to facilitate correspondence (Kellen, 2001, p. 3). Audio files can be uploaded to music sharing sites where it can be downloaded without raising suspicion; audio can also be uploaded to websites and set to automatically play upon visiting the page. The file would then automatically be downloaded to the computer's temporary files. These means would make it hard to tell if the user was deliberately looking for that file (Kellen, 2001, p. 4).

Some argue against digital steganography being a terrorism threat, pointing out the lack of evidence and claiming that low tech steganography (not involving embedded files within other files) could be used instead. In February of 2001, a USA Today article was released claiming that terrorists had been using steganographic methods in addition to encryption to communicate (Kelley, 2001). These claims were found to be false according to Maura Conway (2003, p. 57). Conway claims that low tech steganography is much easier to use and just as effective for secret communication: such as encryption, code words and symbols (2003, p. 56-57). Conway argues that the development of steganalysis and methods available to disrupt communication through steganography deters terrorists from using it (2003, p. 56). In a study seeking to find how much steganographic content is available on the internet, Niels Provos developed a detection program and analyzed over 2 million photos on eBay. No embedded files were detected (Provost & Honeyman, 2002 , p. 12).

Although there is no current evidence to prove that terrorists have been using steganography, it doesn't rule out the possibility. There are many files on the internet and it would be impractical to attempt to examine them all. There are many ways to use digital steganography which makes it easy to bypass current security measures. Also, most of the previous research mentioned is more than ten years old. This research may be outdated now, given the fast-evolving nature of computer technology.

#### Method/Procedure

In my proposal, I stated I would write two separate programs: one to embed a text file into an image file and a second program to un-embed the text file. The goal was to successfully embed a text file size of 100 bytes into a lossless image format (such as a bitmap or PNG) using an LSB (least significant bit) insertion method. Measuring the success of the programs would involve surveying voluntary respondents to determine whether differences in the before and after images were perceivable by the human eye. It was determined that a survey of this type does not involve human subjects research and IRB exemption was granted to this project (see Appendix A).

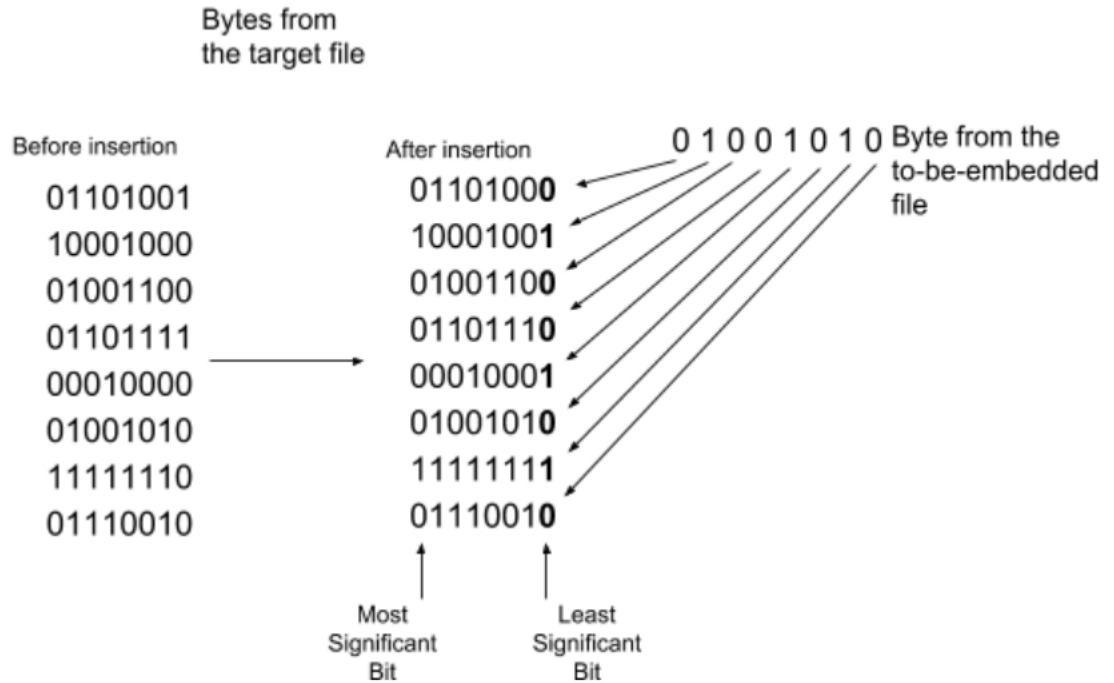


Figure 2: Visual illustration of the concept of LSB insertion

Steganography using LSB insertion is a method of embedding files inside other files by inserting the to-be-embedded file one bit at a time into the least significant bit place of every byte in the target file (see Figure 2). The target file is the file that will hold the to-be-embedded file. The least significant bit is the bit that, when changed, has the least effect on the file. Therefore, removing these bits and replacing them with the bits that make up the to-be-embedded file will alter the target file the least. The goal is to alter the file in a way that is imperceptible to the human eye.

In order to isolate and replace the LSBs in the target file with the bits of the to-be-embedded file, a masking process was used. Masking refers to the process of using

bitwise operators to isolate and alter the data at the bit level. The bitwise operators used in this project were & (AND), | (OR), << (left shift), and >> (right shift).

Operator	Function	Example
& (bitwise AND)	Replaces bit with a 1 if both the original bit and the mask bit are 1. Replaces with a 0 if one or both bits are 0.	<pre> 00101011 &amp; 00000001 *00000001  *Note using &amp; with 1 (00000001 in binary) clears all of the bits except for the rightmost one.</pre>
(bitwise OR)	Replaces bit with a 1 if either the original bit or the mask bit are 1. Replaces with a 0 only if both bits are 0.	<pre> 00101011   00000001 *00101011  *Note using   with 1 retains all of the bits except for the rightmost one.</pre>
<< n (left shift)	Shifts bits left n places. Bits shifted off the left side are discarded, and zeroes are appended to the right.	10101011 << 2 = 101011 <b>00</b>
>> n (right shift)	Shifts bits right n places. Bits shifted off the right are discarded, and zeroes are appended to the left.	10101011 >> 2 = <b>00</b> 101010

Table 1: Bitwise operator functions in C++

First, each byte in the target file is masked with 0xFE, using the bitwise AND operator. 0xFE is the hex notation of 254. In binary, this number is represented as 11111110. This enables the last bit in the byte to be “selected” and keeps the other 7 bits the same. Once masked with 0xFE, bitwise AND is used to mask each byte of the to-be-embedded file with 1 (00000001). This clears all the bits, except for the rightmost one. Therefore, in order to “select” the first bit of the to-be-embedded file, the result of shifting the byte to the right 7 times must be masked with 1. Taking the result of this and the result of the target file byte being masked with 0xFE and using the bitwise OR

operator to mask those together, results in a new byte that contains the first 7 bits of the target file byte and the first bit of the to-be-embedded file bit in the LSB place. Then, the original byte in the target file is set to this new byte to replace the original with the new byte. In C++ code, this looks like the following (see Appendix B for full source code):

```
targetByte = (targetByte & 0xFE) | (to-be-embeddedByte >> 7) & 0x1
```

Once the first byte is set, we can shift the to-be-embedded byte to the left once in order to allow the second bit in the byte to be embedded in the next byte of the target file. This is repeated 8 times to embed the full to-be-embedded byte into the first 8 bytes of the target file. Then, the entire process is repeated to embed the next to-be-embedded byte into the next 8 bytes of the target file. This is repeated until the entire to-be-embedded file is embedded into the target file.

Since this program deals with PNG images, it does not alter each byte in the image, but rather each pixel in the image. Each pixel is made up of 4 bytes: one to hold each red, green, blue, and transparency value for that pixel. The algorithm developed in this project alters the LSB of only 1 byte for each pixel and alternates between the red, green, and blue bytes each time. The transparent byte is skipped since altering a pixel that is transparent when viewing the image may cause a visible alteration. This means that for the first pixel in the image, the LSB of the byte holding the pixel's red value is altered.

Then, in the next pixel, the LSB of the green byte is altered. Then, the blue byte is altered in the following pixel, returning back to the red in the fourth pixel.

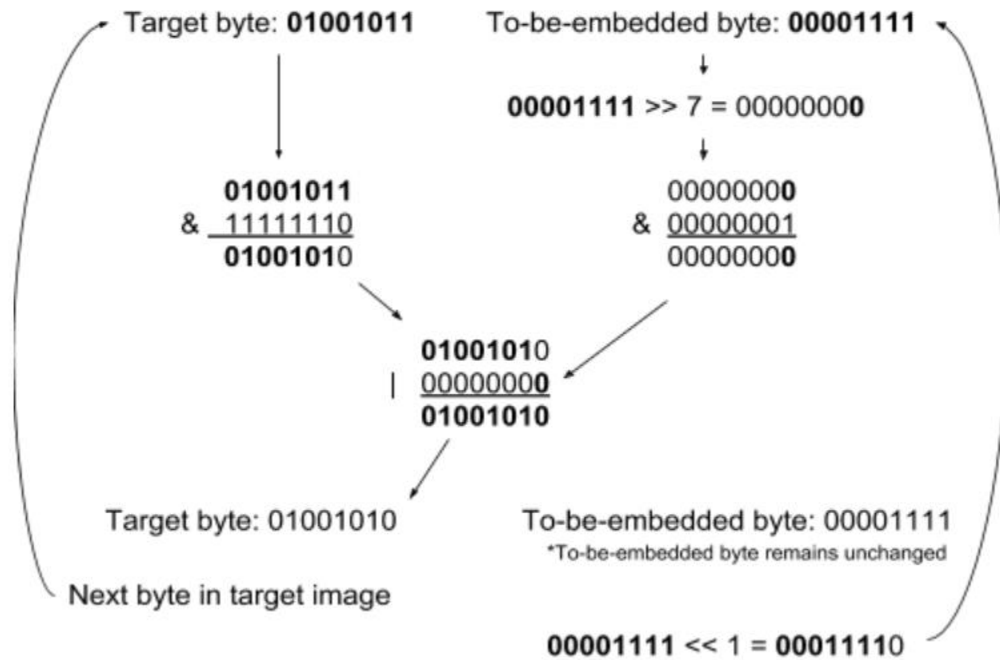


Figure 3: Visual illustration of LSB insertion being implemented in C++

### Product Produced

The product produced was one program with three functions: an embed function, an un-embed function (called “get” when running the program), and a function that takes a PNG image file and calculates the approximate largest size a file can be to be successfully embedded in the PNG (called “getSize”). The embed and un-embed functions were condensed into one program rather than two for easier and faster usage by the end user. The program also contains a “usage” function which displays a short menu



that shows the user what to type in the command line to correctly run the program.<sup>1</sup> This acts as a sort of “quick guide” on how to run the program. The program also contains a “help” function which displays a full manual of all of the commands the program has and how to use them.

The final program was able to embed files in any size PNG file. The embedded file can be any size up to 1/8th of the size of the PNG. The program is only compatible with PNG files, however, any file type can be embedded within the PNG. The following shows a 207 KB (kilobyte) PNG image before and after inserting a 20.7 KB text file within it:

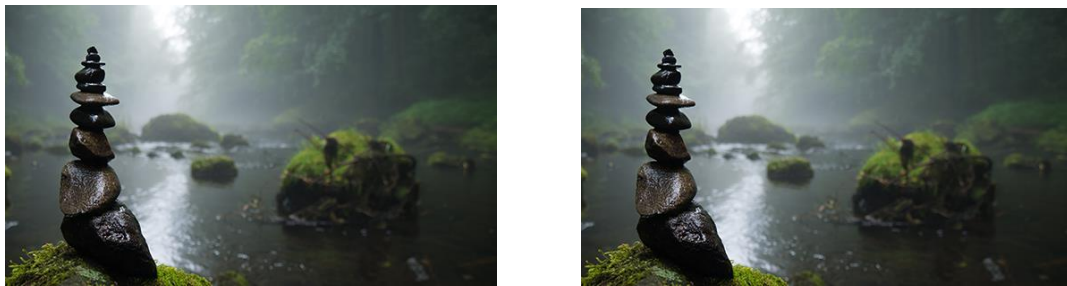


Figure 4: Before (left) and after (right) inserting a text file

Below is a 1.78 MB PNG image before and after inserting an executable file (the Windows equivalent is a .exe file) within it:

---

<sup>1</sup> The command line refers to the window built into all operating systems that allows the user to type in the actions they want the computer to perform (as opposed to using the graphical interface which allows users to click on icons to run and use programs).



Figure 5: Before (left) and after (right) inserting an executable file

Next, is the same PNG image before and after inserting the middle picture (92.1 KB PNG):



Figure 6: Before (left) and after (right) inserting a picture (middle)

The embed function of the program accepts a PNG file and a second file of the user's choice and attempts to embed the second file into the PNG. This is done by using a library<sup>2</sup> called LodePNG (copyrighted by Lode Vandevenne and cited in the program) to load the PNG in a format compatible with the program. The program then reads the second file in binary mode and temporarily stores it in the computer's memory. The size of the second file is also stored and appended to the beginning of the data in memory.

This is done to ensure that only the message is read during un-embedding and extra bits

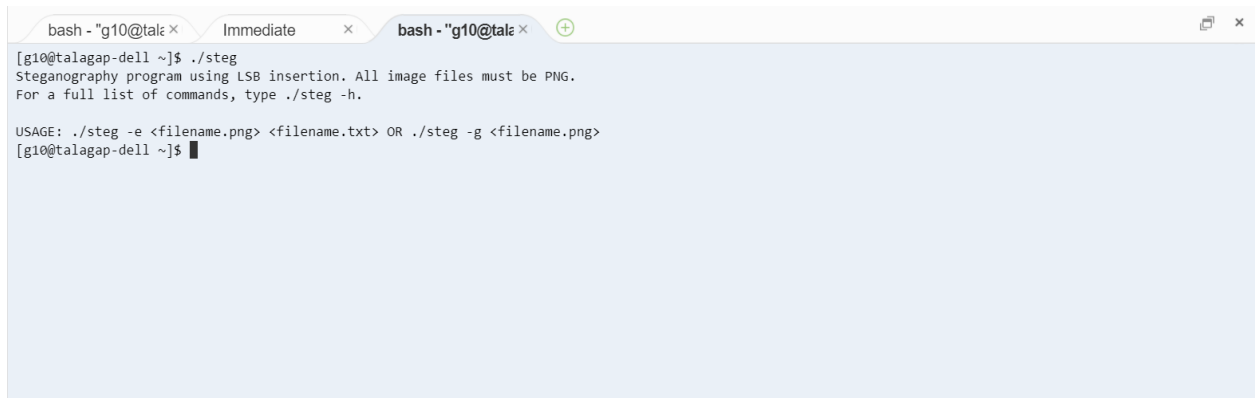
---

<sup>2</sup> A library refers to existing code that programmers share to save other programmers time so that they don't have to code their own versions of the same tasks. This prevents redundancy and allows uniformity. This is similar to how an English dictionary contains already existing words for people to use rather than each person making up their own words to communicate. This allows a common language to help people better understand each other.

that do not contain the message are ignored. The stored file is then embedded into the PNG and a new PNG file is saved. If the total bits of the file with the size appended to the beginning is greater than 1/8th of the size of the PNG, the user is notified that the file is too large to be embedded in the PNG and the program stops. If this is not the case, the LodePNG library is used again to properly save the PNG file. The new PNG file contains the embedded file. Finally, the file stored in memory is deleted to avoid a memory leak. This does not remove the original file saved onto the computer's hard drive but removes the temporary copy of it that was made in the computer's memory. A memory leak occurs when information is not properly deleted from the computer's memory, resulting in more and more memory being used the longer a program is run, which affects the computer's performance.

The un-embed function runs in the same way as the embed function, except it performs the embedding algorithm in reverse in order to extract the embedded file bits in the correct order. The function accepts one file (the PNG with the embedded file) and the extracted bits are saved to the computer's memory. Then, a new file is created and the embedded file's bits are written to that file. First, the size of the embedded file is read (which was appended to the beginning in the embed function) and only that many bytes are written to the new file. This prevents "garbage" values from being saved to the file, which would occur in the case that the bits of the embedded file do not fill all of the LSBs in the PNG (the un-embed function reads every LSB in the PNG file). The new file is identical to the original file before embedding it. Finally, the stream of LSBs from the PNG file contained in the computer's memory are deleted.

The “getSize” function accepts a PNG and reads it, then calculates 1/8th of its size. It then tells the user that they can embed a file up to this amount within the image. Below are images displaying the “usage” and “help” menus that the user is able to access when running the program:



```

bash - "g10@talagap-dell ~" x Immediate x bash - "g10@talagap-dell ~" x
[g10@talagap-dell ~]$ ./steg
Steganography program using LSB insertion. All image files must be PNG.
For a full list of commands, type ./steg -h.

USAGE: ./steg -e <filename.png> <filename.txt> OR ./steg -g <filename.png>
[g10@talagap-dell ~]$

```

Figure 7: Usage menu



```

bash - "g10@talagap-dell ~" x Immediate x bash - "g10@talagap-dell ~" x
[g10@talagap-dell ~]$ ./steg -h

MANUAL:
-s: -size Calculates approx. number of bytes that can be stored in the file. ./steg -s <filename.png>
-e: -embed Embeds the message file in the PNG file. Saves a separate output file. ./steg -e <filename.png> <filename.txt>
-g: -get Reads the message embedded in the file and saves to a separate output file. ./steg -g <filename.png>

Pulls LSB from every byte in the image, alternating RGB values.
[g10@talagap-dell ~]$

```

Figure 8: Help menu

## Results

To test the success of my program, we created a survey displaying 25 images with “before” and “after” versions side by side (see Appendix C). Images were randomly chosen to be altered using my program while the remainder were left unaltered.

Respondents were asked to determine whether the before and after images were “different” or the “same”. Whether the images were altered or unaltered was not disclosed to the respondents.

The survey was made available on Facebook and Twitter and respondents consisted of colleagues who voluntarily responded to the survey. There were 33 total respondents. The following table illustrates the survey’s results.

Image	Correct Answer	# of Correct Responses	# of Incorrect Responses	% of Correct Responses
1	DIFFERENT	11	22	33.33%
2	DIFFERENT	31	2	93.94%
3	SAME	23	10	69.70%
4	SAME	22	11	66.67%
5	SAME	24	9	73.73%
6	DIFFERENT	7	26	21.21%
7	SAME	23	10	69.70%
8	DIFFERENT	6	27	18.18%
9	DIFFERENT	12	21	36.36%
10	SAME	23	10	69.70%
11	SAME	17	16	51.52%
12	SAME	26	7	78.79%
13	DIFFERENT	11	22	33.33%
14	SAME	22	11	66.67%
15	SAME	17	16	51.52%
16	SAME	17	16	51.52%
17	DIFFERENT	8	25	24.24%
18	DIFFERENT	7	26	21.21%
19	DIFFERENT	16	17	48.48%
20	DIFFERENT	4	29	12.12%
21	SAME	19	14	57.58%
22	DIFFERENT	6	27	18.18%
23	SAME	24	9	72.73%
24	SAME	20	13	60.61%
25	SAME	29	4	87.88%
			<b>Average:</b>	51.52%

<b>Total # of pictures that were different:</b>	11
<b>Average correct “different” guess:</b>	32.78%
<b>Total # of pictures that were the same:</b>	14
<b>Average correct “same” guess:</b>	66.23%

Table 2: Survey responses

On average, respondents correctly guessed whether the images were the same or different 51.52% of the time, suggesting that they most likely made random guesses on which images they thought were different and which they thought were the same.

Respondents were more likely to correctly identify images that were unchanged than images that were changed. Of the images that were the same, the percentages of correct responses (identifying that the before and after versions of the image were unchanged) were all above 50%. Of the images that were different, the percentages of correct responses were all below 50%, with the exception of Image 2 (93.94%). On average, the percentage of respondents who correctly identified images that were actually different was 32.78%. The average percentage of respondents who correctly identified images that were the same was 62.23%. This further supports the conclusion that respondents were guessing whether the images were the same or different, rather than actually seeing a visible difference. The fact that there were more unchanged pictures than changed pictures may have skewed these results.

Image 2 was correctly pointed out to be different by 93.94% of the respondents, suggesting that there was an actual visible difference between the before and after image and that respondents did not randomly guess whether its before and after versions were changed or unchanged. We believe this was the case because Image 2 was the only black and white picture included in our sample. This makes sense because the algorithm used affects the RGB values in images. It can be concluded that my program is less inconspicuous, and therefore less effective, when the images used are not color images.

### Analysis/Conclusion

The findings suggest that answers to the survey were chosen randomly with respondents not being able to see an actual difference between the images. The chance of correctly identifying an altered image was just over 50%. The findings also suggest that it was easier to identify when images were unchanged rather than changed.

Respondents may have been biased since the survey explained its purpose at the beginning of the questionnaire. Respondents may have thought they saw differences that weren't actually there. Conversely, they may have been expecting to not see any differences and therefore didn't examine the images as closely.

Another possible source of inaccuracy was the clarity of the instructions. One respondent mentioned to me that they could not "find any hidden messages," to which I then explained that they wouldn't be able to see the message, but a slight difference in color or noise in the image.

Since survey responses were anonymous, which were submitted through Google Forms, there was no way to tell if any respondents did the survey multiple times.

In conclusion, I consider the program to be effective and in compliance with best practices of steganography for color images. Further improvements will need to be made to extend its effectiveness to black and white images.

Although the focus of this project was on accurate implementation of steganography and not the accuracy of the statistical analysis of the survey results, there are some improvements that could be made in a later project to provide more insight regarding the survey. Other aspects of the project that have room for improvement are the

conditions with which the effectiveness of the program was tested. A larger sample size of respondents and an equal number of unchanged and changed images would have given more accurate results. More statistical testing, rather than the descriptive analysis I have already given, would also provide more insight into whether differences in the images are visible or not.

Future research could explore the differences my algorithm created between black and white images. Changing my algorithm to expand to each RGB value in each byte of the image, rather than alternating between them every 3 bytes, may have produced a more uniform distribution of bits, creating difference results. Future research could also be expanded to test methods other than the one created in this project, as well as methods other than LSB insertion.

### Reflection

This project was implemented to demonstrate that secret communication could go undetected using steganography. During the project, I learned that there are more potential uses for steganography, such as infecting victims with malware. I was able to successfully embed executable files in images, demonstrating the viability of the current research and the ease with which it can be implemented. With additional code that triggers the embedded executable to run, this could be implemented as a cyber weapon. This method of malware distribution potentially bypasses security measures of websites that are currently seen as secure. For example, Gmail blocks executable files from being sent as email attachments, however it allows image files. PNG image files with



executables embedded in them would only be recognized as a PNG and would not be blocked.

As confirmed in the process of writing my embed and un-embed functions, the only way to correctly pull the message out of the file is to un-embed it in the exact same way that the file was embedded in it. This confirms that it is very possible that steganography has been used for terrorism-related communication, corroborating the existing research in this field of study. Currently, there is plenty of opportunity on the Internet to post pictures that seem inconspicuous, and since the exact method used to embed hidden files would have to be used to pull an intelligible message from it, communication can go undetected with ease. Further, there are many existing methods of embedding files available, as well as more than can be created.

On an individual level, this project served to expand my knowledge of programming and the makeup of files at the bit and byte level. I also developed a much deeper understanding of C++, which was the programming language used to write this program, and a deeper understanding of binary. Throughout the process of writing my program, I developed stronger problem-solving skills, as well as how to go through the debugging process when programming. I also developed a deeper understanding of steganography and the potential uses and threats it poses. The skills and knowledge I gained through the process are valuable and necessary to a career in cyber security.

The project demonstrates the viability of steganography as a potential threat to cyber security, the most prominent threat being the difficulty with detecting the usage of steganography. Currently, the only methods we have for detecting steganography are

“brute force” methods: trying different methods of unembedding until one works, or unsuccessfully detecting embedded files. In cases where the original image is available, the original and suspected images can be tested for differences, however, in a real-world situation, the likelihood of having the original image available is low.

Prevention is more likely to be the solution to reducing the threat of steganography. Organizations can compress their files to reduce redundancy, which would remove any embedded files before sharing them (Wiseman, 2018). However, this would only protect against data being leaked from organizations or malware getting into organizations’ networks. This would have no effect on detecting or preventing secret communication between terrorist groups.

Analysis of the existing steganography software available on the Internet could possibly be a step in the direction of developing a viable detection method. Figuring out the common methods these programs use and creating a database of known steganography methods would create a greater likelihood of detection.

## References

- Conway, M. (2003). Code Wars: Steganography, Signals, Intelligence, and Terrorism. *Knowledge, Technology & Policy*, 16(2), 45-62.
- Kellen, T. (2001). Hiding in Plain View: Could Steganography be a Terrorist Tool? *SANS Institute InfoSec Reading Room*, 1-7. Retrieved from Homeland Security Digital Library
- Kelley, J. (2001, Feb. 5). Terror Groups Hide Behind Web Encryption. *USA Today*. Retrieved from <http://usatoday30.usatoday.com/tech/news/2001-02-05-binladen.htm>
- McMillen, Dave. (2017, Nov. 16). *Steganography: A Safe Haven for Malware*. Retrieved from <https://securityintelligence.com/steganography-a-safe-haven-for-malware/>
- Murphy, S. (2004). Steganography--The New Intelligence Threat. *Marine Corps University*, 1-10. Retrieved from Homeland Security Digital Library
- Nosrati, M., Karimi, R., & Hariri, M. (2011). An Introduction to Steganography Methods. *World Applied Programming*, 1(3), 191-195.
- Provos, N., & Honeyman, P. (2002). Detecting Steganographic Content on the Internet. *Center for Information Technology Integration. University of Michigan*. 1-13  
<<http://www.citi.umich.edu/u/provos/papers/detecting.pdf>>
- Vandevenne, Lode. (2017). LodePNG. [Computer software library]. Retrieved from <http://lodev.org/lodepng/>
- Villinger, S. (2011, May 9). Crash Course: Digital Steganography. *IT World*. Retrieved Nov. 21, 2016 from <http://www.itworld.com/article/2826840/crash-course-digital-steganography.html>
- Wang, H., & Wang, S. (2004). Cyber Warfare: Steganography vs. Steganalysis. *Communications of the ACM*, 47(10), 76-82.
- Wiseman, Simon. (2018, Feb. 6). *What You Need to Know about Stegware*. Retrieved from <http://www.idgconnect.com/blog-abstract/29346/what-stegware>
- Zielińska, E., Mazurczyk, W., & Szczypi, K. (2014). Trends in Steganography. *Communications of the ACM*, 57(3), 86-95.

Appendices

Appendix A: IRB Exemption Letter



Cheri Walker-Owens <walkerowensc@uindy.edu>

---

**Question about IRB approval**

---

**Greg Manship** <manshipg@uindy.edu>  
To: Cheri Walker-Owens <walkerowensc@uindy.edu>  
Cc: James Williams <williamsjb@uindy.edu>

Fri, May 26, 2017 at 2:42 PM

Afternoon, Cheri.

I am profoundly sorry for my lack of communication on this matter. I have no excuses or explanations, other than I forgot to follow up because your project is not human subjects research. Nevertheless, I write this email to document closure of our correspondence.

This email confirms that I have read the copy of your Honors College proposal that you attached to your previous email, and I have determined that your Honors College project does NOT satisfy the federal regulatory definitional criteria of "human subjects research" because you are not collecting information about individuals when interacting (i.e., collecting questionnaire information) with them. Hence, because your project is not human subjects research your project is not within the purview of the Human Research Protections Program (HRPP). Therefore, UIndy will not require human research protections review (i.e., exemption determination) of your project prior to implementation.

I have copied Dr. Williams on this email in order to apprise him.

I invite you to follow up with questions or concerns.

Greg

[Quoted text hidden]  
[Quoted text hidden]

Director, Human Research Protections Program (HRPP)  
University of Indianapolis  
Health Pavilion, Room 261  
1400 East Hanna Avenue  
Indianapolis, IN, 46227

[317/781-5774](tel:3177815774) (Office)  
[317/788-5299](tel:3177885299) (Fax)

## Appendix B: Project Source Code

```

// steg.cpp
// author: Cheri Walker-Owens
// Bug & error fixes written by Paul Talaga
// uses LodePNG version 20170917 Copyright (c) 2005-2017 Lode
Vandevenne
// date: February 16, 2018
// TO COMPILE: g++ -o steg steg.cpp
// TO RUN: ./steg

#include <iostream>
#include <string>
#include <iomanip>
#include <fstream>
#include <vector>
#include "lodepng.cpp"

using namespace std;

void usage();
void help();
void getSize(char*[]);
void embedFile(char*[]);
void getFile(char*[]);

int main(int argc, char *argv[]){
    if(argc == 1){
        usage();
    }

    else if(argc == 2 && string(argv[1]) == "-h"){
        help();
    }

    else if(argc == 3 && string(argv[1]) == "-s"){
        getSize(argv);
    }

    else if(argc == 3 && string(argv[1]) == "-g"){
        getFile(argv);
    }

    else if(argc == 4 && string(argv[1]) == "-e"){
        embedFile(argv);
    }

    else{
        usage();
    }

    return 0;
}

```

```

void usage(){
    cout << "Steganography program using LSB insertion. All image files
must be PNG." << endl;
    cout << "For a full list of commands, type ./steg -h." << endl;
    cout << "\nUSAGE: ./steg -e <filename.png> <filename.txt> OR ./steg
-g <filename.png>" << endl;
}

void help(){
    cout << "\nMANUAL:" << endl;
    cout << "-s: -size Calculates approx. number of bytes that can be
stored in the file. ./steg -s <filename.png>" << endl;
    cout << "-e: -embed Embeds the message file in the PNG file. Saves
a separate output file. ./steg -e <filename.png> <filename.txt>" <<
endl;
    cout << "-g: -get Reads the message embedded in the file and
saves to a separate output file. ./steg -g <filename.png>" << endl;
    cout << "\n Pulls LSB from every byte in the image, alternating RGB
values." << endl;
}

void getSize(char *argv[]){
// Opening PNG file
    unsigned width, height;
    vector<unsigned char> image;
    unsigned error = lodepng::decode(image, width, height, argv[2]);
    if(error)cout << "Error! " << lodepng_error_text(error) << endl;

// Calculating approximate number of bytes that can be stored in the
image
    int length = width * height / 8 / 4;

    cout << "Approximately " << length << " bytes can be stored in this
image." << endl;
}

void embedFile(char *argv[]){

    string output_filename = "embedded-" + string(argv[2]);

// Opening and reading PNG file, uses lodepng.cpp & lodepng.h
    unsigned width, height;
    vector<unsigned char> image;
    unsigned error = lodepng::decode(image, width, height, argv[2]);
    if(error)cout << "Error! " << lodepng_error_text(error) << endl;

// Reading message file & storing file size in the beginning of the
message
    // Modified version of code taken from
http://www.cplusplus.com/doc/tutorial/files/
    string input_file = string(argv[3]);
    streampos size;
    char *memblock;

```

```

ifstream file (input_file.c_str(), ios::in|ios::binary|ios::ate);
if (file.is_open()){
    size = file.tellg();
    memblock = new char [int(size) + 4];
    file.seekg (0, ios::beg);
    file.read (memblock + 4, size);
    file.close();
}

int filesize = int(size);
// PGT changed this
char* message = memblock;
memcpy((void*)memblock, &filesize, 4);
filesize = filesize + 4; // To account for the int on the
beginning.

// Embedding message file in PNG file
int image_pos = 0;
int length = width * height / 8 / 4;
unsigned c = 0; // color, either 0,1,2

if(filesize > length){
    cout << argv[3] << " is too large to be embedded into " <<
argv[2] << "." << endl;
    exit(1);
}

for(int i = 0; i < length && i < filesize; i++){
    for(int b = 0; b < 8; b++){
        unsigned x = image_pos % width;
        unsigned y = image_pos / width;
        image[(y * width + x) * 4 + (image_pos % 3)] =
(image[(y * width + x) * 4 + (image_pos % 3)] & 0xFE) | (message[i] >>
7) & 0x1;
        image_pos++;
        message[i] = message[i] << 1;
    }
}

// Writing new PNG file with embedded message file
error = lodepng::encode(output_filename.c_str(), image, width,
height);
if(error)cout << "Error! " << lodepng_error_text(error) << endl;

delete[] memblock;
}

void getFile(char *argv[]){
// Opening PNG file
unsigned width, height;
vector<unsigned char> image;
unsigned error = lodepng::decode(image, width, height, argv[2]);

```

```

    if(error)cout << "Error! " << lodepng_error_text(error) << endl;

// Un-embedding message
// Reads LSB from entire PNG
int image_pos = 0;
int length = width * height / 8;
unsigned c = 0; // color, either 0,1,2
char* out_text = new char[length];

for(int i = 0; i < length; i++){
    char o = 0;
    for(int b = 0; b < 8; b++){
        unsigned x = image_pos % width;
        unsigned y = image_pos / width;
        o = o << 1;
        o = (o & 0xFE) | ((image[(y * width + x) * 4 + (image_pos %
3)]) & 1);
        image_pos++;
    }
    out_text[i] = o;
}

// Reads length of message stored at the beginning of message and
writes message_length # of bytes to file
int message_length;
memcpy(&message_length, (void*)out_text, 4);
ofstream decryptedMessage;
decryptedMessage.open("decryptedMessage.txt", ios::binary);
//decryptedMessage << out_text;
decryptedMessage.write(out_text + 4, message_length);
decryptedMessage.close();

delete[] out_text;
}

```



## Appendix C: Survey

## Image Steganography Survey

Image Steganography is the method of embedding a secret message inside an image. To evaluate the effectiveness of our implementation, we'd like you to attempt to recognize a modified image, which has a message embedded inside of it. The colors are modified to embed the message, so you may be able to discern a difference between the original image (left) and the modified one (right).

Below are pairs of images. The LEFT image is the original image, and the right may have a message embedded, or not. Your task is to identify the pairs of images which look DIFFERENT and which are the SAME, below each image.

\* Required



Image 01 \*

☐ DIFFERENT

☐ SAME



Image 02\*

☐ DIFFERENT

☐ SAME



Image 03 \*

O DIFFERENT

O SAME



Image 04 \*

O DIFFERENT

O SAME



Image 05 \*

O DIFFERENT

O SAME





Image 06 \*

☐ DIFFERENT

☐ SAME

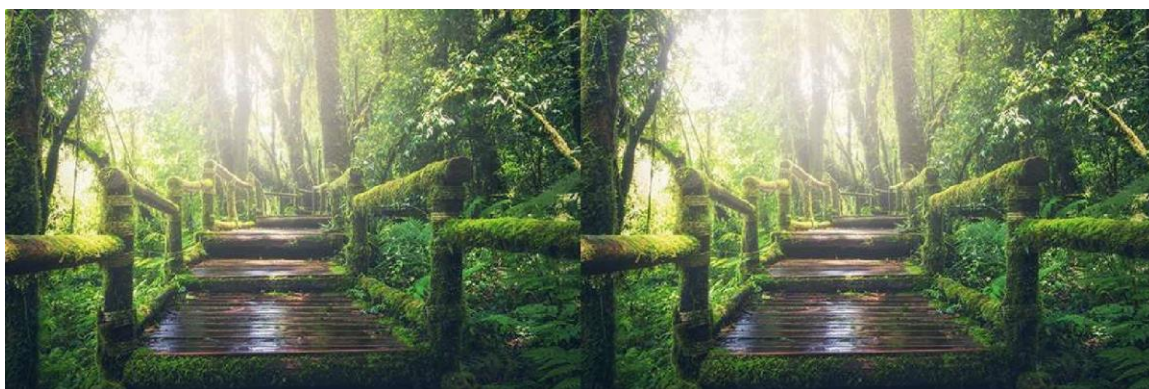


Image 07 \*

☐ DIFFERENT

☐ SAME



Image 08 \*

☐ DIFFERENT

☐ SAME



Image 09 \*

☐ DIFFERENT

☐ SAME



Image 10 \*

☐ DIFFERENT

☐ SAME



Image 11 \*

☐ DIFFERENT

☐ SAME





Image 12\*

☐ DIFFERENT

☐ SAME



Image 13 \*

☐ DIFFERENT

☐ SAME



Image 14\*

☐ DIFFERENT

☐ SAME



Image 15\*

☐ DIFFERENT

☐ SAME



Image 16 \*

☐ DIFFERENT

☐ SAME



Image 17\*

☐ DIFFERENT

☐ SAME





Image 18 \*

O DIFFERENT

O SAME



Image 19 \*

O DIFFERENT

O SAME

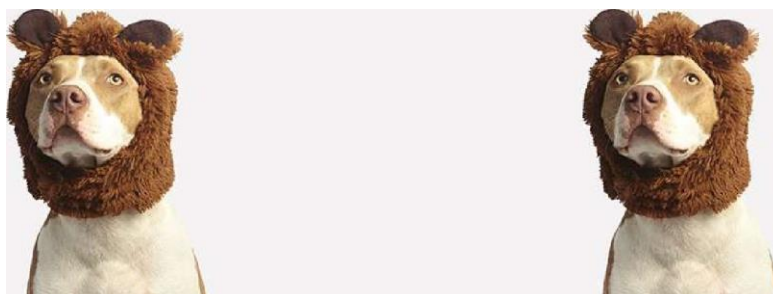


Image 20 \*

O DIFFERENT

O SAME



Image 21 \*

☐ DIFFERENT

☐ SAME



Image 22 \*

☐ DIFFERENT

☐ SAME



Image 23 \*

☐ DIFFERENT

☐ SAME





Image 24

☐ DIFFERENT

☐ SAME



Image 25

☐ DIFFERENT

☐ SAME

SUBMIT