“Waiter! There’s a Message in My Soup!”

Uncovering Steganography

Adventures in Supercomputing Challenge
Final Report
April 2, 2003

Team #022
Bosque School

Team Members
Samuel R. Ashmore
Jessica E. Behles
A. Zoe Dennis

Teachers
Debra M. Loftin
Dorothy I. Ashmore

Project Advisor
Christopher Nebergall

Table of Contents

List of Figures

List of Tables

Executive Summary

Following the September 11th attacks, people became increasingly worried about the possibility of additional terrorist attacks. It was obvious to officials that these attacks were very coordinated, which required an extreme amount of communication – communication that occurred right under American noses. But how could such communication pass unnoticed by the entire world? The answer, according to a July 10, 2002 *USA TODAY* article, is steganography.

Steganography is the process of using a computer program to hide messages in types of files such as MP3s or GIFs. Files doctored by steganography programs look and sound no different to human senses than any other file – the only difference lies deep within the files coding. The *USA TODAY* article describes situations in which terrorist groups are using popular internet sites such as eBay to pass messages using steganography.

In response to this threat, our project was to write a program that could remove any hidden information from an image. Because that would have been a project far beyond our capabilities, we limited our project to a program that can remove steganography from GIFs. Another discovery that changed our project was watermarks – copyright information imbedded into a file using steganography. This kind of steganography cannot be removed because it is illegal to tamper with copyright information.

The program we wrote is capable of detecting and removing steganography that has been imbedded by two different programs from GIF images. In addition it is capable of recognizing watermarks and leaving them intact.

Introduction

The Problem

On July 10, 2002, *USA TODAY* struck fear into the hearts of terrorist-stricken Americans by publishing an article stating that terrorist groups such as al-Qaeda are using the Internet to spread pro-terrorist propaganda as well as information about attacks on the United States – like the September 11th attacks. The article said these "militant groups" were hiding their messages in images using a method known as steganography -- the art of hiding a message within a medium, images or sound files for instance, to be extracted at its destination. Hidden within the actual bits of the file, the message is very difficult for humans – and even computers -- to perceive. According to *USA TODAY*, these groups are using steganography in images on the popular auction site eBay, as well as adult websites and newsgroups.

A week later, *Salon.com* published an article stating that the *USA TODAY* article was far-fetched, as eBay was contacted neither by the author of the article nor by the FBI. Following the article, eBay was searched, but no such hidden messages were found. The *Salon.com* article also said, however, that just because such files remain undetected does not mean that they do not exist; only that the terrorists' steganography technology may be greater than our detection technology. If that is the case, the fear that terrorists are using steganography is quite valid and can be seen as a very real threat. Steganography is difficult to detect, often encrypted, and steganography tools free and very easy to obtain via the Internet. So, what is to be done about the threat of terror groups using steganography?

The Solution

Our original solution to this problem was to create a computer program that is capable of detecting and decoding a message that has been hidden in an image using steganography. Unfortunately, such a project is very expensive and time-consuming. Another problem is that such messages, in addition to being hidden with steganography, are often encoded with cryptography, making deciphering the message much harder. Because of the impracticality of this idea, our project's new focus was to look at the problem from a different angle.

The new aim of our project was to write a computer program that can destroy information hidden in an image file without degrading the image's visible quality – a process known as "scrubbing." We had originally picked JPEGs to work with; however, they are much less practical for steganography due to their compression algorithm. Because of this, we decided to use GIF images instead. When our program scrubs the GIF file, it effectively destroys the hidden information, eliminating any need to decrypt it. Additionally, our program is capable of distinguishing between signatures left by different steganography programs. This ability is important because copyright information (or a watermark) is frequently stored in a file using a method similar to steganography, known as watermarking. Our program can determine if the hidden information is a watermark. If it is, then the program will not scrub it.

Our program has several possible applications on computers today. For example, businesses could use it to prevent employees from using steganography to leak confidential information. This process would be a simple matter of scanning all the

images that are sent from company computers and accounts. Another use for our program would be for large commercial Internet-based companies, such as eBay. With the threat of terrorists using images on the site to pass information, additional security measures must be taken to incapacitate this method of passing hidden messages. Using our program, these companies, could easily eliminate any information hidden in images. Additionally, this program is very appropriate for a supercomputing project. While the program itself is relatively simple, the number of images on the Internet is constantly growing, and all of these could be harboring concealed messages, resulting in an equally large number of images that need to be scanned for steganography. When used with a supercomputer, our program is capable of dividing tasks across several processors, causing a dramatic increase in the number of images that can be processed in a given time. On a site like eBay, where thousands of new images are displayed each day, increased processing speed is essential.

Background Information

Historical Steganography

Steganography comes from Greek and means, literally, "covered writing." Any method of transferring undetected information is known as steganography. This usually involves hiding a message in some medium, so that it passes under the eyes of authorities unnoticed. Its purpose is to pass a message without detection, and if there is a perceivable hint that something is there, it is not true steganography. This is different from cryptography in that cryptography simply scrambles a message while steganography completely hides the message from view.

The first recorded use of steganography was by commanders in Ancient Greece. When a message needed to be sent to another city, yet remain a secret, the commanders would resort to steganography. The head of a Greek soldier would be shaved, and the message would be tattooed to his head. He would then allow his hair to grow out, completely hiding the message. Then he would travel to the recipient's city, with the message remaining completely undetected, and drawing no suspicion. At his destination, he would shave his head, revealing the message.

A later form of steganography was used in Ancient Rome. In order to do this, the Romans utilized their most common form of written communication: wax-covered tablets. A wooden tablet would be covered in wax, and then a message would be carved in the wax. To send a hidden message, a messenger would scrape the wax from the tablet, and then carve the message directly into the wood. Then he would again cover the tablet in wax, giving it the appearance of an unused tablet. Sometimes,

the messenger would carve an additional message into the wax layer. While appearing to be a blank tablet, or even some innocuous message, the tablet would pass without suspicion. The recipient would retrieve the message by scraping off the wax.

More recently, even into the 20th century, steganography has involved methods such as invisible inks. Some liquids such as milk, urine or lemon juice dry clear, but darken when heated. These "inks" would be used to write a message on a letter – again utilizing the most common form of written communication. The letter would be sent using regular mail, passing completely undetected by censors and other such authorities. When the letter was delivered, the hidden message could then be revealed with heat or sometimes special chemicals. This method was frequently used during World War II. Other recent methods include placing tiny dots or holes beneath certain characters in a letter, spelling out a hidden message, and also using dots and dashes of letters such as i's and t's to spell out messages in Morse code. By using the mail system, people again utilized a common form of communication for steganography.

Modern Steganography

In this age of computers and the Internet, steganography has moved into the digital realm. An earlier form of digital steganography, known as random dot stereograms, has actually been used very little for passing messages. Random dot stereograms use a computer program to hide an image by generating random dots. When correctly focused upon, the dots will reveal the hidden image. Although stereograms have not been widely used, they have been made famous by the *Magic Eye* book series. These books contain many images hidden using stereograms;

however, these only require a certain, relatively simple method of visual focusing to reveal. These hidden images are easily discovered, and further, are meant to be found. As such, stereograms do not fit into the true definition of steganography: a hidden message that is completely concealed from perception.

Although the above examples are types of steganography, the form of steganography that we are using in our project is digital, meaning that its use is limited to computers. In digital steganography, the hidden message is frequently referred to as a "package," or "payload." The "package" is hidden in a medium known as a "cover" or "envelope." A cover is normally a file such as a GIF, MP3, WAV, or JPEG. In addition, a method was recently developed to hide a message in .exe (Executable) files. The most common and easily available steganography programs are for image files (e.g. GIFs). An attempt to detect, uncover, or decode steganography is known as an "attack." Detecting steganography means to discover its existence; "scrubbing" means to remove or simply destroy the package. "Steganalysis" is the practice of attacking steganography. In using steganography on the Internet, people today are yet again utilizing their most common form of written communication. *Figure 1* shows an example of this digital steganography. The message, "If only it was this easy to detect!!!" was inserted into the image using a steganography program. No difference is visible between the cover before the information was inserted, and after.

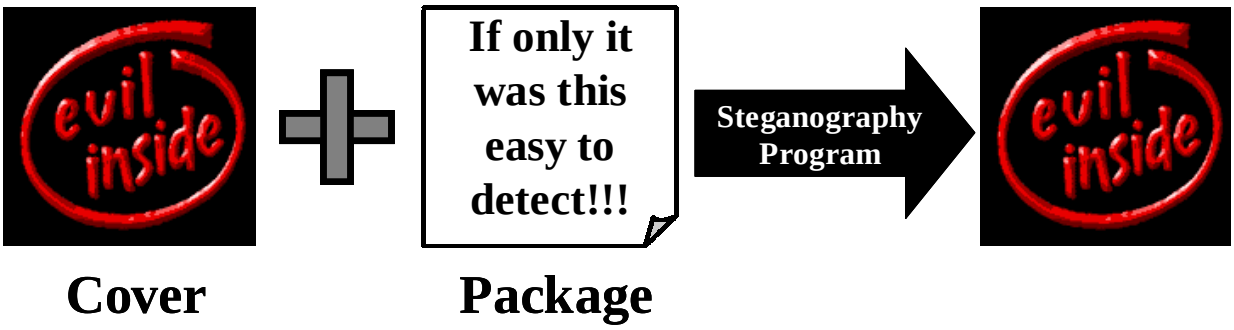

Figure 1. Message Hidden Using Gifshuffle

The best images for hiding information are 'busy' images, or images with many different contrasting colors. The more contrast there is between different pixels, the less likely that the steganography can be seen with the human eye. For example, if an image with a single, solid color has information inserted, it will be very easy to see the difference between the original cover and the final cover. On the other hand, if an image has many different pixels with contrasting colors, the steganography will be impossible to see.

In image files, the information is hidden in either the least significant bits (LSBs) or the color palette of the image. These parts affect – and degrade -- the image itself very little, decreasing the chance that the change can be detected. Additionally, steganography is often used with encryption. With this method, a message is encrypted before it is hidden in the cover file. This makes it even more difficult to retrieve the file, since decrypting the file adds an additional step to the process. This complication makes the need to destroy the package even more pronounced; if authorities cannot read the message, they need to destroy the message before it reaches its destination.

Types of Images and Programs

On the Internet, the two most common image types are GIF and JPEG. JPEGs use lossy compression, which means that the data for the exact image is not stored but the computer develops an equation for its compression that approximates the original image and color values, but is not exact. This form of compression results in changes, or even losses, in the least significant bits of the pixels. Steganography programs hide information in these equations. Because of this form of estimating compression, steganography can easily be destroyed by simply opening and saving the image.

The other image type, GIFs, contains and stores the exact color-values of the image. The GIF compression algorithm is non-lossy, so it does not estimate the values when it compresses the data. Because of these exact values, it is more difficult to destroy hidden data. In this image type, the data can be stored in either the least significant bits of the pixels, or the color-values in the color palette. Programs that hide information in the least significant bits of the pixels are called LSB-hiding programs. Programs of this type include The Third Eye, Hide and Seek, and S-Tools. Programs that hide information in the palette of the image are known as palette-hiding programs. Gifshuffle is a palette-hiding program.

Steganalysis

Steganalysis is the practice of detecting steganography. It undermines the basis of steganography – the principle of remaining undetected – by virtue of simply detecting it. Most steganalysis programs use some kind of statistical analysis to determine the likelihood of the presence of hidden information. The X^2 (or Chi-Squared)

test is one such test, which assumes that there is a hidden message. This works by comparing the image's pixel arrangement to pixel placements observed in previous stego-images. If these values match, then the possibility of a hidden message exists. Another form of steganalysis is the "known cover" attack, where there is access to the cover image before the package has been inserted. In this method, the images from before and after steganography processing are compared. Our program uses a "steg-only" form of attack, where only the after-processing image is known.

Project Description

The Original Plan

The original aim of our project was to create a program that was capable of destroying any hidden information in an image, eliminating the need to detect beforehand, regardless of the type of information hidden in the file. We had to change this goal for several reasons. The first reason was that there are two different ways to hide information in an image. We had to take both of these methods into account in our program. We also modified our goal because the original project was not comprehensive enough for an advanced supercomputing project. The final reason we changed was that we found out about watermarks: information hidden inside a file to denote ownership or copyrights. This discovery fueled our project in a major way. Because we did not want to create a program that was capable of removing copyrights, we had to expand it so that it could recognize watermarks and cease processing the image.

We had also originally planned to focus on JPEG images. We decided against that direction because the lossy compression algorithm makes hiding and maintaining a message in a JPEG extremely difficult. The ease of message removal in this image type made our project essentially obsolete for JPEGs. As a result, we turned our focus to GIFs – another very common image type on the Internet. These images are very susceptible to information hiding, and steganography tools to accomplish this are easily acquired.

Our Program

An image has two places to hide information: in the least significant bits (LSBs), and in the palette. Our program is capable of detecting and removing both types of hiding. We have also chosen seperate a steganography program to use for each of these hiding techniques. For the palette-hiding, we decided to use Gifshuffle. We chose this program because it was the only open-source palette-hiding program. For LSB-hiding, we chose The Third Eye because it was a freeware program that was supposed to become open-source. It never became open, however, but by the time this was discovered, we had done too much work on the project to go back and restart with a different program.

Palette-Hiding

Of the two methods, palette-hiding is much easier to remove. In a GIF, the palette is a numbered index of colors that enables the file to be compressed. It does this by replacing the color values with index values, resulting in an image size that is in direct relation to the number of indexed colors. In most normal cases, these values are ordered in one of two ways: by the natural order and by luminance. The natural order is where all of the colors are given a new number, known as the order value. This value is found using the equation [65536 * the color's red value + 256 * the color's green value + the color's blue value]. When all the colors are given order values, they are then sorted in ascending order. The sorting by luminance, or brightness, is similar, differing only in the equations used. With luminance, the order value equation is [the color's red value*.299 + the green value * .587 + the blue value * .114].

A palette-hiding program, such as Gifshuffle, inserts information into an image by arranging the colors in their natural order, and then swaps pairs of these color values according to the inserted message's binary. If two colors in the palette are swapped, it represents a 1. If there is no color swap, a 0 is represented. *Table 1* gives a simple example of how this method works. In addition to swapping pairs of colors, information is also hidden in the palette by generating new colors and duplicates of colors already in the palette. The processes that Gifshuffle uses to hide information are shown in *Figure 2.*

	Original Order			First Swap			Second Swap			Third Swap		
Palette Color Values	R	G	B	R	G	B	R	G	B	R	G	B
	0	0	0	0	0	0	64	64	64	64	64	64
	64	64	64	128	128	128	0	0	0	128	128	128
	128	128	128	64	64	64	128	128	128	0	0	0
Message Binary Values	None (00)			01			10			11		

Table 1. Example of Palette-Hiding

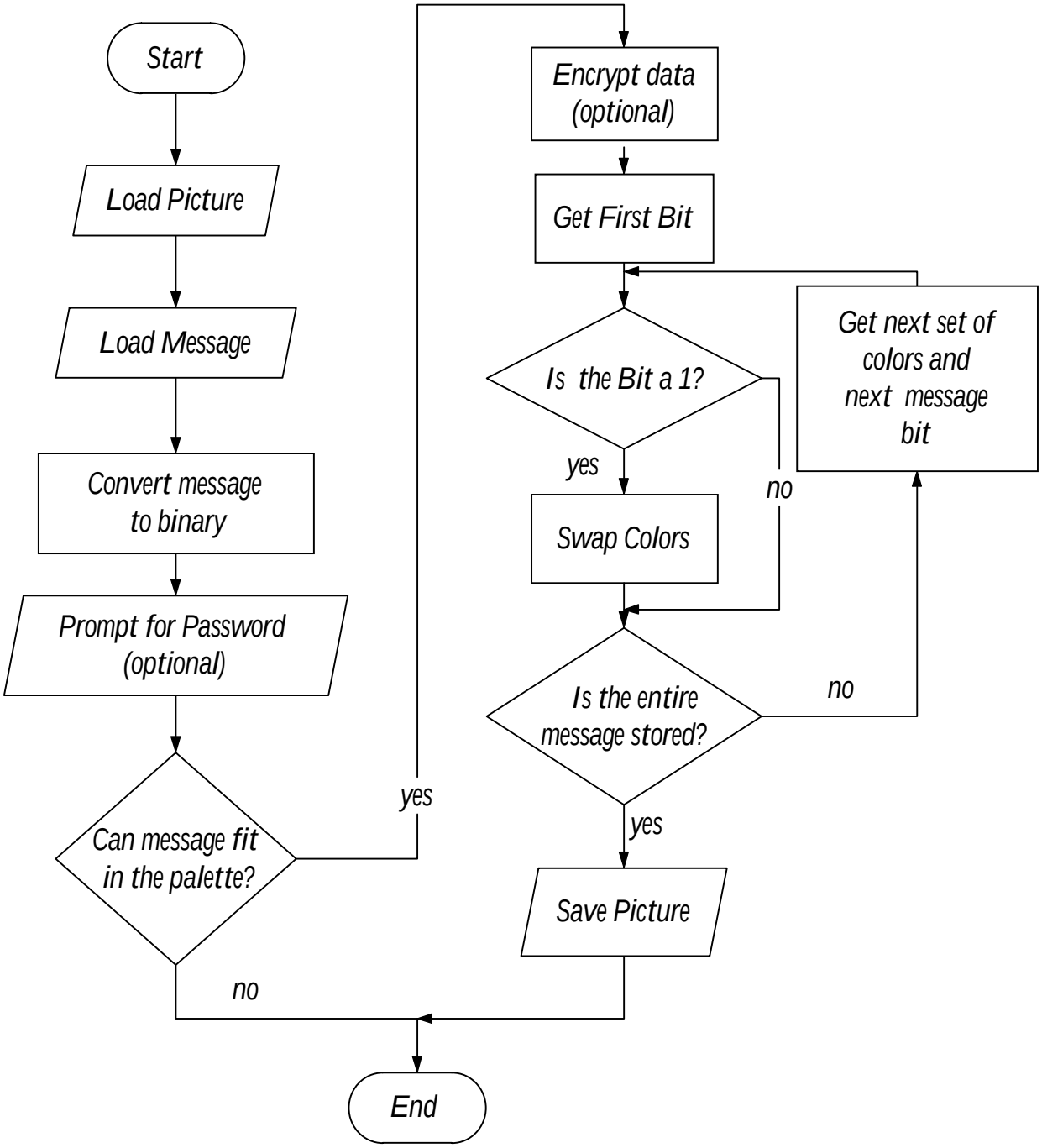

Figure 2. Gifshuffle Palette-Hiding Algorithm

Palette-Scrubbing

The process of detecting and removing information hidden in a GIF's palette is the first step our program takes towards scrubbing an image. It does this by extracting

the image's palette using CxImage, a series of C++ functions and classes that allows images to be opened, processed and saved. After extracting the palette, the program then checks for either natural or luminance ordering, and checks for duplicate colors. If the palette is correctly ordered, and there are no duplicates, the palette is ignored, and the program moves on to the next step. If it is not already ordered, or duplicate colors are discovered, then the program counts the number of colors in the palette. If the total is less than 216 colors, it begins to re-sort the palette. This step is necessary because the web palette consists of 216 colors, making it very difficult for hiding information. If it is more than 216 colors, the program extracts all of the color values in the image and uses them to create a new palette, which is indexed and reordered into the natural order. The program then reduces the palette by removing duplicates of colors. This new, ordered palette replaces the image's original palette, deleting any information hidden in the original. The scrubbed palette is then returned to the image, and the program moves on to the next process. *Figure 3* is a simplified illustration of this process.

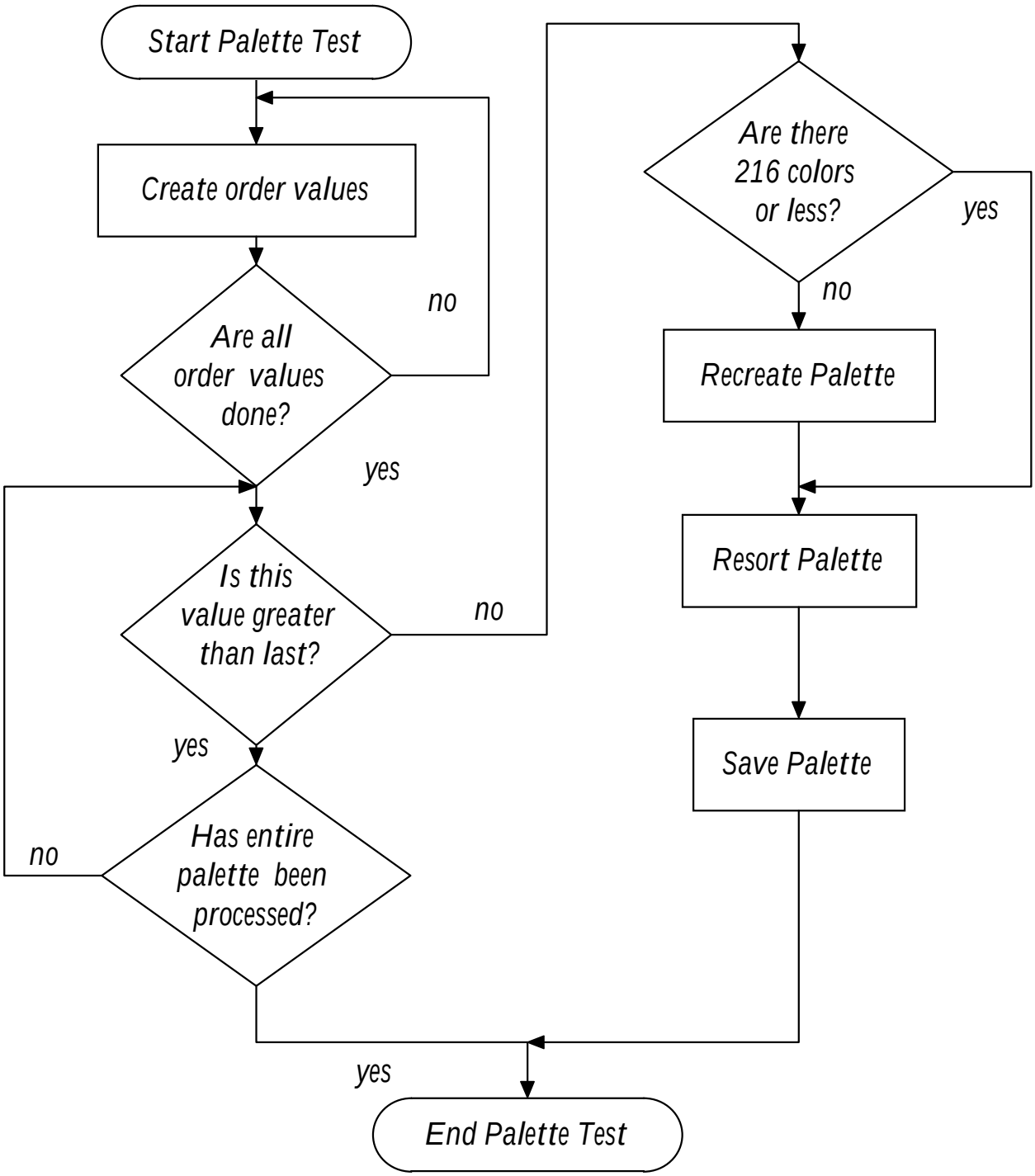

Figure 3. Palette Detection/Scrubbing Algorithm

Despite easy detection by a computer, palette sorting has some definite advantages. Editing the image, using techniques such as cropping, resizing, rotating,

and many other similar image conversions, does not destroy the palette's hidden information. This differs from LSB-hiding, in which image conversions can easily destroy hidden information. In addition, if another color is added to the palette, most image editors will not completely re-sort the palette but merely add the new color to the end of the index. This color-adding does not compromise the message since it leaves the palette in the same order.

Another strength is that a program must look specifically in the palette for the message to be noticed, since not all image editors allow people to view this palette. In addition, changing the palette has absolutely no degradation on the image and sometimes even raises the quality. Also, GIF images have a palette for their compression. This allows small images, even sized one pixel by one pixel, potentially to store a larger message than other methods allow. The problem with palette sorting is that changes occur in the palette order, leaving hidden information relatively easy to detect if the palette can be accessed. Another weakness of this method of hiding is that the amount of information that can be hidden in the palette is limited. Because there are only 256 red values, 256 green values, and 256 blue values, the maximum amount that can be stored in the palette is 209 bytes, or about 200 ASCII characters. While the method very good since it is difficult for average users to detect, palette-hiding is used much less frequently because it is very easy to remove, and also because of its size limitations. Since palette-hiding is so easy to remove we decided it should be the program's first step.

LSB-Hiding

The second option for hiding information in an image is in the LSBs. Each pixel of an image has three color values: red, green and blue. For the purposes of steganography, these values are expressed in binary. For example (11111111 11111111 11111111) is white, where each group of eight bits represents one of the RGB values (in this case, white, 255 255 255). Information can be stored and hidden in up to three bits of each value without making changes perceivable by humans. The bits changed in this fashion are always the lower (last) three of each value. When information is inserted, its binary is hidden in the color values of the pixels. *Table 2* gives an example of this kind of hiding, inserting the binary for the ASCII character 'B' into three pixels, and illustrates how the RGB values change very little. *Figure 4* shows the algorithm that The Third Eye uses to insert information into an image.

	Binary Values	RGB Values
Original 3 Pixels	(11111111 11111111 11111111) (10000000 10000000 10000000) (00000000 00000000 00000000)	(255 255 255) (128 128 128) (000 000 000)
3 Pixels With 'B' (01000010) Inserted (Underlines denote changes to bits)	(1111111<u>0</u> 11111111 1111111<u>0</u>) (10000000 10000000 10000000) (0000000<u>1</u> 00000000 00000000)	(254 255 254) (128 128 128) (001 000 000)

Table 2. Example of LSB-Hiding

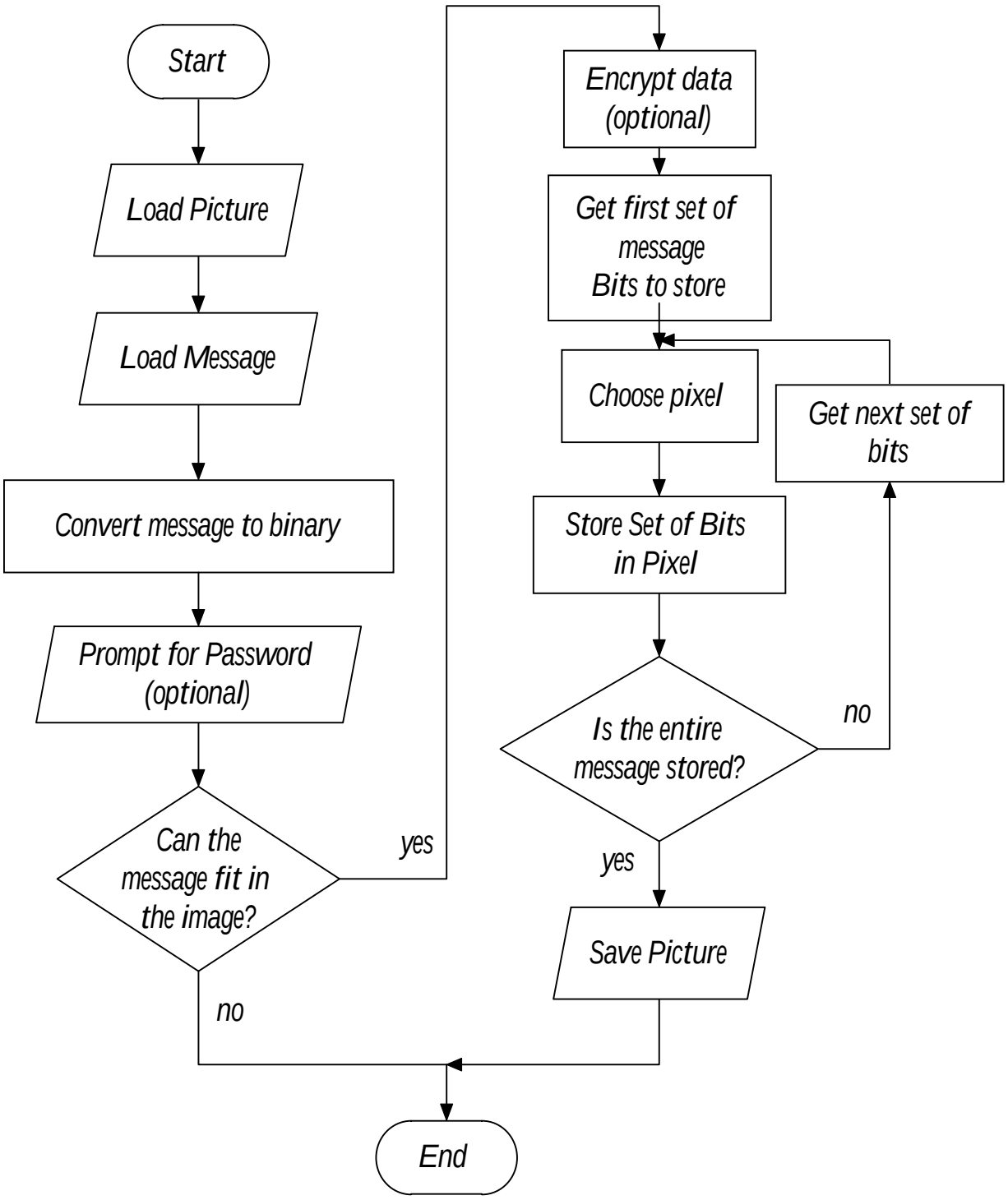

Figure 4. The Third Eye LSB-Hiding Algorithm

Signatures

The aspect that makes the LSB-scrubbing part of our program possible is what we call a signature. When a steganography program inserts information, it will use the same method for every image it processes. Examples of this insertion include changing pixels in a certain pattern that is unique to a single program, such as every tenth pixel, or by distorting colors in a certain way, (for instance, changing green values in a way that no other program utilizes). In addition, a program may also insert header information, which contains information about the program used, into the image before the actual package is added. Using these patterns of pixel changing, and these headers, we can define a signature: a list of characteristics in the hidden information, that is specific to a single steganography program. For example, 'Program A' hides information in every 38^{th} pixel and changes red values by two. If we find an image where information is hidden in every 38^{th} pixel, and all red values are changed by two, we know that we have an image that was processed by Program A.

In order to develop these signatures, we compared images using the "known cover" attack. To do this we inserted a package into a cover using a certain steganography program. By comparing the cover before insertion to the cover following insertion, we were able to see exactly what methods the program had used to change the LSBs in the image. When we knew these insertion characteristics, we were able to identify what program was used based on how the image was inserted.

A fully developed signature has three characteristics defined. The first is a recognizable pattern of information storage (e.g. where and how far apart bits of

information are stored in the image). The second definition is whether a header is hidden along with the package. If there is, the size is noted. The final characteristic is how each color is distorted. Using these three characteristics, we have complete signatures that our program is capable of recognizing.

Signatures are important because when our program processes an image's LSBs, it determines which steganography program was used. This determination allows our program to clean the changed (or infected) bits correctly, according to what program was used to hide the package. Another reason that signatures are a necessity is so that the program can recognize whether the hidden information is a watermark. Because watermarks contain ownership and copyright information, removing them is illegal. According to the Digital Millennium Copyright Act of 1998, "Making or selling devices or services that are used to circumvent either category of technological measure [of copyright] is prohibited" (4). Thus, we had to write a code that was capable of recognizing and avoiding watermarks. Signatures allow our program to recognize if the steganography program was a watermarking program. When a signature from a watermarking program is recognized, our program will pass over that image, leaving the watermark intact. We used a watermark program called ReaWatermark.

LSB-Scrubbing

The LSB-scrubbing part of our program also uses CxImage to open and analyze images. The first step our program takes for scrubbing the LSBs is to load all of the defined signatures. It then searches the image for any color distortions, possible header information, and patterns in locations and range of color distortions based on the signatures that we have developed. When it finishes this task, the program will compare

possible distortions with each signature. If the number of similarities is smaller than the size of the header, the program will move on to the next signature. The reason for this is that a steganography program will always make the size of the information inserted larger than, or equal to, the size of the header. If all the signatures are cycled through with no signature matches, program moves on to the next image. However, if the number of similarities was smaller than the header size, the program takes this result as not finding any matches, and will continue processing.

If the program recognized the signature as coming from a watermarking program, it will stop processing the image. On the other hand, if the package is not a watermark, the program will begin the scrubbing stage. By previously identifying what program was used to insert the message, our program now knows which pixels have been changed. Using this information, it locates the infected pixels. For each infected pixel, the program averages the values of the four surrounding pixels. This average value replaces those of the infected pixel, removing the bits that held the hidden message.

Once all infected pixels have been removed, the program changes some extra, uninfected bits in the image. These bits are pseudorandom within certain parameters based on the signatures. The reason for this step is that the recipient of the package can compare it to the original cover, if it is available. Because our program changed the same pixels as the steganography program did, the recipient can locate the message by comparison. By changing some uninfected pixels, the program makes it impossible for the message to be retrieved by comparing the image to the original cover. Once this step is completed, the program saves this new image, and goes on to the next one. *Figure 5* is a simplified illustration of the LSB detecting and scrubbing process.

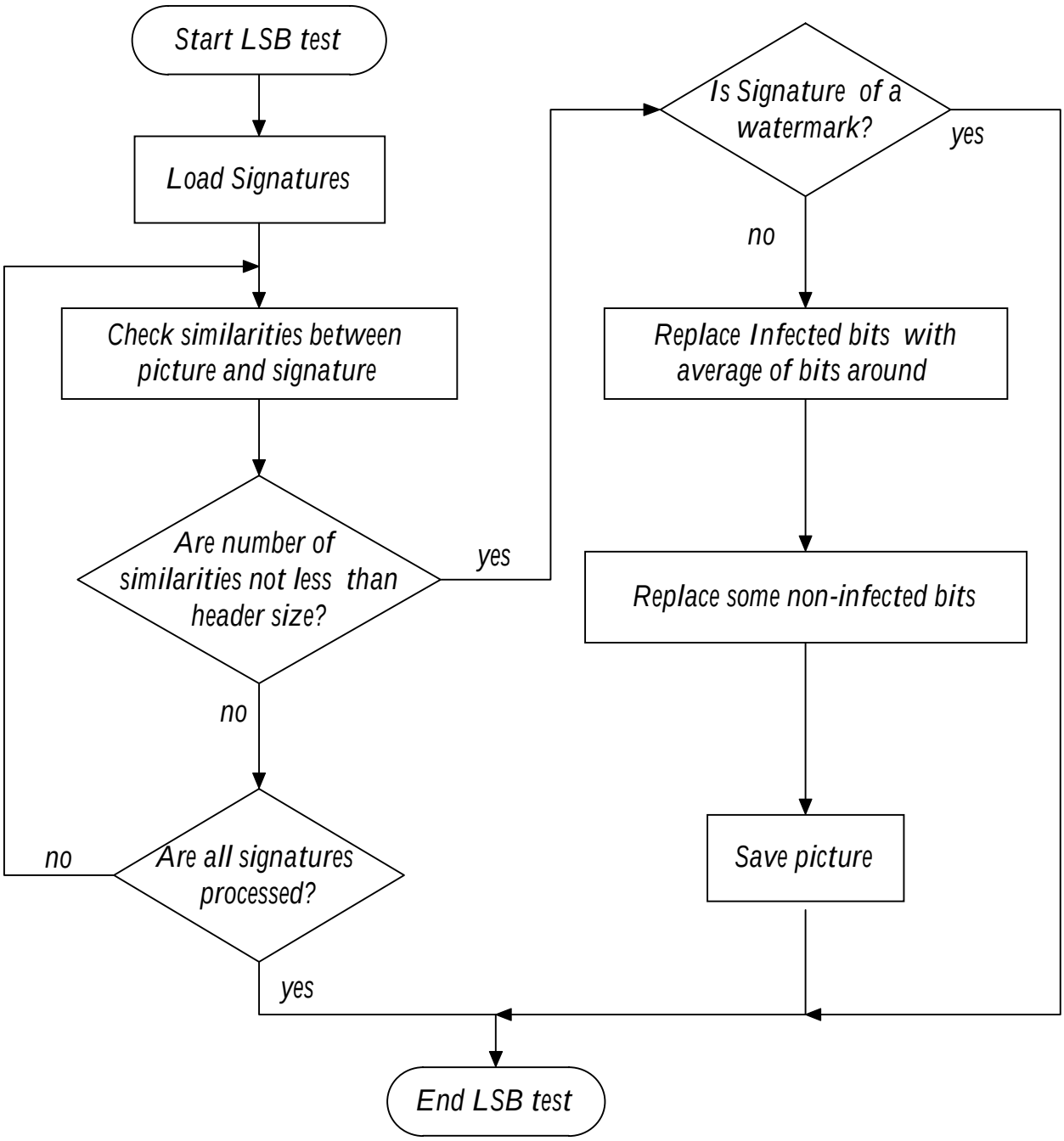

Figure 5. LSB Detection/Scrubbing Algorithm

Information hidden in the least significant bits is much harder for a computer to detect. This process, however degrades, the image – something that palette-hiding does not do. Cropping the image destroys information hidden in the LSBs because this manipulation can remove pixels containing parts of the message. However, when the

image needs to remain unaltered, the removal of messages hidden by LSB-hiding is much more difficult than those hidden by palette-hiding. This difficulty is not only because the alteration is harder to find than in palette-hiding, but also because removing the information degrades the image further. Additionally, watermarks are also hidden in the least significant bits, and they must not be removed. LSB-hiding programs are used much more frequently than palette-hiding programs because LSB-hiding is harder to detect and remove. Another advantage of LSB-hiding is that the LSBs are not limited in storage capacity the way that the palette hiding is. We chose to make the LSB detecting and scrubbing part of the program the final step because of its complexity and importance, since LSB-hiding programs are much more common.

Results

Palette Detection and Removal

Our program produced results for detection and removal for images that consisted of a single color, "solid," and images with many contrasting colors, "busy." In our testing we used 5 different payloads and 7 different images

Solid Images

The processing time for detection in the palette of a solid image (similar to the image in *Figure 6*) was very short because there the program contained very few processes that had to be completed. In terms of accuracy, the program was very good. In our initial testing, It detected every message that was hidden in images, and only found messages where there were none (false-positives) in about one of every twenty runs, this occurred in images that started off with strange palette, These palette where not checked before the testing and could have contained messages.

Figure 6. Solid Image

In our preliminary results, the scrubbing time for solid palettes was longer than that for detection, but still relatively short only a about 20 loops. The program always managed to clean all traces of messages from images, with no image degradation. Additionally, if the image had a palette greater than 216 colors, the program reduced the palette, resulting in a smaller file size.

Busy Images

In our preliminary testing, in busy images, detecting time was very short. All messages were detected, with false-positives in only about one of twenty runs, this is due to the same reason that solid images had. Even though imputed images were busy, the processing speed is not affected, since the complexity has nothing to do with the palette. A very complex image, such as that in *Figure 7*, requires no extra processing time for detection.

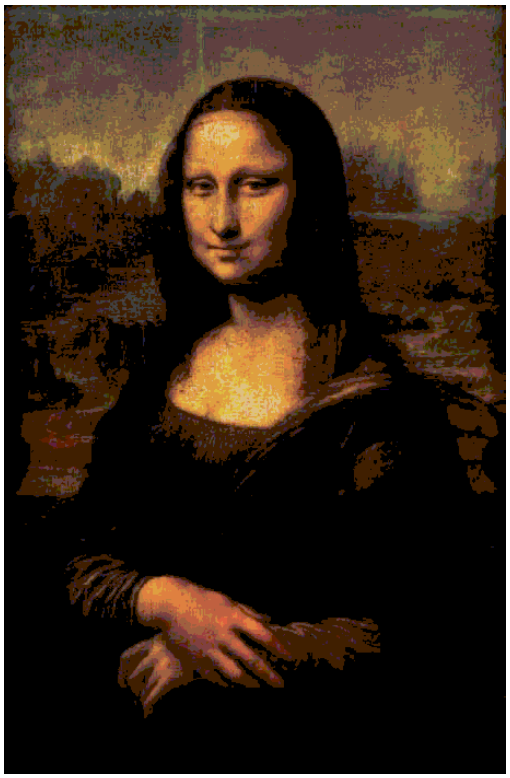

Figure 7. "Busy" Image, The Mona Lisa

In scrubbing, our program successfully cleaned every hidden message with no degradation of images.

LSB Detection and Removal

Our work with the LSB portion of our program is still being refined to make the system much more effective. We need to define more signatures and streamline the process. Our program produced results for detection and removal for images that consisted of a single color, "solid," as well as images with many contrasting colors, "busy."

Solid Images

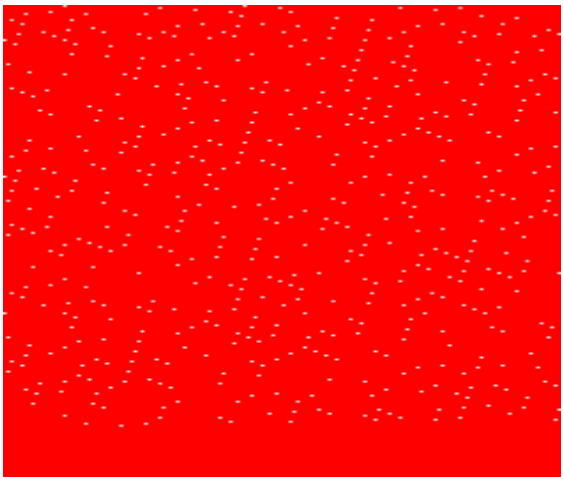

Figure 8. "Solid" Image with data hidden by The Third Eye

Detecting LSBs is a very complicated process because the program has to search so much of the file, load the signatures, and then find matches. Therefore, our program took longer to detect LSB-hidden images than to detect palette-hidden images. . In fact, the amount of time that detecting the LSB-hidden images required was about the same as scrubbing the palette. In solid images no false-positives occurred. *Figure 8* is an example of a solid image that was processed with The Third Eye.

The removal of LSBs required about the same length of time as the detection process. On solid images, the program has a perfect success rate. Every image tested was cleaned of all package information, with all images returning to their original quality before insertion.

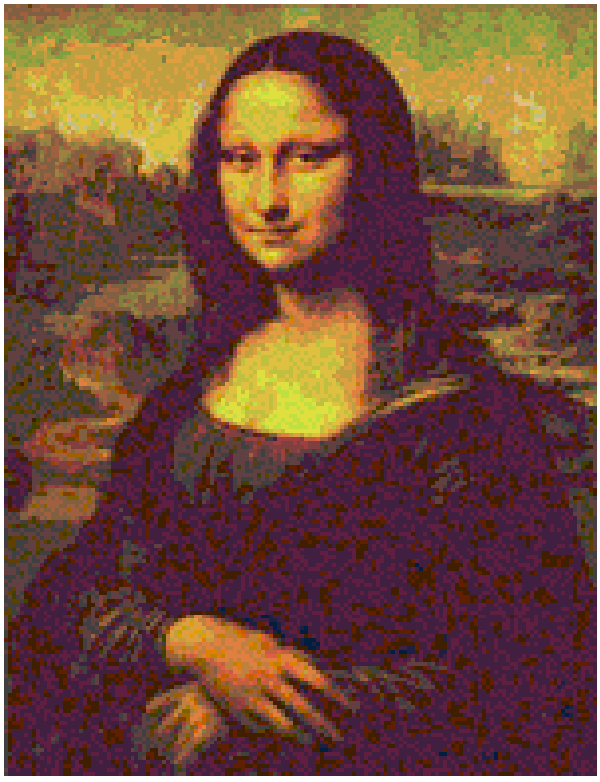

Figure 9. "Busy" Image with information hidden by The Third Eye

Busy Images

For detection, busy images were much harder on the program than solid images, resulting in a longer processing time. The program found most hidden messages in busy images, such as that in *Figure 9*, but missed approximately one in every forty. About one in ten scans resulted in a false-

positive. These errors were most frequently showed up in images with colors resembling those that are connected with a signature.

Scrubbing of LSBs in busy images required somewhat more time than the detection process. The program had very high accuracy, missing about three bits out of every 100. The image was degraded somewhat more than the steganography program degraded it in the first place, but in no cases could this degradation be perceived by the human eye.

Signatures and Watermarks

The signature portion of the LSB program worked amazingly well. The program detected every watermark, and never cleared any by mistake. Additionally, it produced about one false-positive in 15 trials. We started out with 15 images and used a couple different watermark logos and visibilities on each one. The false-positives were caused by images that contained light regions that appeared to consist of a single color but are actually two different colors next to each other.

Conclusions

After running tests on many images, some with messages and some without, we found that the palette-test portion of our program was stronger. However, palette-hiding methods are not used as frequently because they are limited in size and easily removed. Palettes also cannot store watermark information, so when running tests on palettes we did not need to worry about removing watermarks. This is the first piece of code that we were able to run effectively. In hindsight, it is understandable why so few people use palette-hiding for steganography. Palette-hiding is weaker because it is so simple to detect and to destroy the message.

It was more essential that our LSB program worked better since more people use LSB-hiding steganography programs. Images hidden using LSB-hiding programs are much harder to detect, as the message is integrated into the image. Although slight changes such as cropping or altering the image size can destroy the message, removing the message is much harder to do without changing the image. The watermarks also make LSBs difficult, since they cannot legally be removed. We are very glad that we realized the watermark problem when we did. because, not only did it keep us on the right side of the law, but it made our project more complex.

Overall, our program worked satisfactorily, and we are very proud of the results.

Recommendations

Had we had more time to work on this project, there would have been several options for strengthening our program. To begin with, we would like to have been able to work with other programs. While Gifshuffle and The Third Eye are good representatives of their types; all steganography programs insert information differently. In addition, having more LSB-hiding programs for experimentation means that we could have had more signatures for our LSB detection program to analyze.

Another useful enhancement to our program would allow it to continue to scan the image after discovering a watermark. Ideally, the program would leave the watermark intact, but still be able to scan for additional steganography. This would be useful because a steganography program could insert a message into an already watermarked image. As our program is now, a possible loophole lies in the watermarking program. Because our program stops scanning when it discovers a watermark, someone could insert a watermark to avoid the program.

Acknowledgements

Challenge Team 22 would like to thank the following people for their contributions to our project:

Dick Allen for providing us information on the Digital Millennium Act of 1998, which has legal implications for our project.

Dorothy Ashmore for always challenging us to do our best.

Debbie Loftin for setting up the Adventures in Supercomputing Challenge at Bosque, for driving us to Glorieta, and for her continual help with our presentation.

Christopher Nebergall for his advice and support throughout the project as our mentor.

Laurel Behles, Charles Boyer, Gail Lane, and Heather O'Shea for reviewing our final report.

References

El-Khalil, Rakan "Hydan: Information hiding in Program Binaries" 15,March, 2003 <http://www.crazyboy.com/hydan/>

Johnson, Neil F. and Sushil Jajodia. "Exploring Steganography, Seeing the Unseen" IEEE *Computer*, February 1998 < http://www.jjtc.com/pub/r2026.pdf>

Johnson, Neil F., Zoran Duric, and Sushil Jajodia. Information Hiding, Steganography and Watermarking- Attacks and Countermeasures. Boston: Kluwer Academic Publishers, 2001.

Kallen, Ian, and Eric Perlman. "Common Internet File Formats." 19 Dec. 1995. 21 Sep. 2002. http://www.matisse.net/files/formats.html

Korhan, Karen. Steganography uses and effects on Society. 2002 U or Illinois. http://www.cpsr.org/essays/2002/2rr3.html

Kuhn, Markus. "Steganography." 3 Jul. 1995. *IKS*. 21 Sep. 2002. <http://www.iks-jena.de/mitarb/lutz/security/stegano.html>

Lilley, Chris. "JPEG JFIF." 18 Jul. 2002. *World Wide Web Consortium*. 21 Sep. 2002. <http://www.w3.org/Graphics/JPEG/>

Manjoo, Farhad. "The case of the missing code." 17 Jul. 2002. *Salon.com*. 21 Sep. 2002. <http://www.salon.com/tech/feature/2002/07/17/steganography/>

McCullagh, Declan. "Bin Laden: Steganography Master?" 7 Feb. 2001. *Wired News*. 21 Sep. 2002. <http://www.wired.com/news/politics/0,1283,41658,00.html>

Petitcolas, Fabien A. P. "The information hiding homepage: digital watermarking & steganography." 17 Jun. 2002. 21 Sep. 2002. <http://www.cl.cam.ac.uk/~fapp2/steganography/>

Provos, Niels. "Steganography Press Information." Center for Information Technology Integration. 4 Jan. 2002. U of Michigan. 21 Sep. 2002. <http://www.citi.umich.edu/u/provos/stego/faq.html>

Rosenberger, Jackie. "Steganography – Camouflage and Concealment Revisited." *Devwebpro*. 21 Sep. 2002. <http://www.devwebpro.com/2002/0809.html>

Stafan Katzenbeisser, Fabien A. P. Pettitcolas, eds. Information Hiding techniques for steganography and digital watermarking. Massachusetts: Artech House, 2000.

"Stegano." 20 Jun. 2001. *Cameleon.* 21 Sep. 2002.
<http://www.bugbrother.com/www.cameleon.org/1stegano.html>

"Steganography." 12 Apr. 2002. *Webopedia.com*. 21 Sept. 2002.
<http://www.webopedia.com/TERM/S/steganography.htmlv>

"Steganography: The Science of Hiding Information." Detroit Now Internet Advisor. 20 Jun. 2002. WJR, Detroit. 21 Sep. 2002.
<http://www.internetadvisor.net/week6-20-02.html>

"Steganography." 2001. U of Michigan. 21 Sep. 2002.
<http://www.citi.umich.edu/u/provos/stego/>

Wayner, Peter. Disappearing Cryptography, Information Hiding: stenganography & watermarking 2nd ed. San Diego: Morgan Kaufmann Publisher, 2000.

U.S. Copyright Office The Digital Millennium Copyright Act of 1998, U.S. Copyright Office Summary. December 1998.

Appendix A: Software/Steganography Tools

CxImage is not a steganography tool. It is a C++ image library, made up of functions and class designed to deal with a variety of different images. This library allowed us to open, edit, view, and save images. A steganography tool could easily be created using this image library. CxImage can be found at http://www.codeproject.com/bitmap/cximage.asp

Gifshuffle is an open-source program that can be found at http://www.darkside.com.au/gifshuffle/ . This software was the only palette-hiding open-source program that we were able to find on the Internet.

The Third Eye is a freeware program that can be downloaded from http://www.webkclub.com/tte/. The software was originally supposed to become open-source. It did not, however, but we had progressed to far on the project to begin anew with a different program. This program is a LSB-hiding program.

The final tool that we used was ReaWatermark by ReaSoft. This tool allows people to put copyright information and watermarks on images. It can be found at http://www.reasoft.com/. There is a trial version that lasts for 30 days, which is what we used to create our program's signatures.

Appendix B: Code

This is our palette-detection and -scrubbing code. This is the simplest program that we ran over the course of our project. We have chosen not to include our LSB code because of legal problems involving the watermark detection part of our program. The Digital Millennium Copyright Act of 1998 clearly states that removing a watermark is illegal. "Making or selling devices or services that are used to circumvent either category of technological measure [of copyright] is prohibited…."(The Digital Millennium Copyright Act of 1998). By simply removing the watermark detection portion of our program, a person could potentially use our program to remove watermarks from images.

If you would like to see the LSB portion of our code, please contact us through Samuel Ashmore at ch022sra@mode.lanl.k12.nm.us

```
//
// Palette.cpp
// Team 022
//

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <complex.h>
#include "./CxImage/ximage.h"

//
//File Extension
//Purpose:
//Finds the extension of a file so that the
//decoding can occur
//
void FindExtension(const char *name, char **ext)
{
```

```
    int len = strlen(name);

    for (int i = len - 1; i >= 0; i--) {
        if (name[i] == '.') {
            *ext = (char *) (name + i + 1);
            return;
        }
    }

    *ext = (char *) (name + len);
    return;
}

//
//Image Type
//Purpose:
//Figures which type of image is inputted
//
int inline imageType(char *extin)
{
    int typein;

    if (strcmp(extin, "bmp") == 0)
        typein = CXIMAGE_FORMAT_BMP;
    else if (strcmp(extin, "gif") == 0)
        typein = CXIMAGE_FORMAT_GIF;
    else if (strcmp(extin, "ico") == 0)
        typein = CXIMAGE_FORMAT_ICO;
    else if (strcmp(extin, "tga") == 0)
        typein = CXIMAGE_FORMAT_TGA;
    else if (strcmp(extin, "jpg") == 0) {
        cout << "Error does not Support steganography in jpegs.\n"
            << "Data stored in compresion and can easily be \n"
            << "destroyed by openning the file for processing\n";
        typein = CXIMAGE_FORMAT_JPG;
    }
    else if (strcmp(extin, "tif") == 0 || strcmp(extin, "tiff") == 0)
        typein = CXIMAGE_FORMAT_TIF;
    else if (strcmp(extin, "png") == 0)
        typein = CXIMAGE_FORMAT_PNG;
    else if (strcmp(extin, "wbmp") == 0)
        typein = CXIMAGE_FORMAT_BMP;
    else if (strcmp(extin, "pcx") == 0)
        typein = CXIMAGE_FORMAT_PCX;
    else {
        return -1;
    }

    return typein;
}

//
//Palette Check
//Purpose:
//Checks the order of the palette
//Order: Natural Order and Luminance Order
//
```

```
int palettecheck(RGBQUAD * pal, int palsize)
{
    RGBQUAD *test = pal;
    int rgbvalues[palsize][3];
    int sortvalue[palsize];     //Natural Order
    float sortvalue2[palsize];  //Luminance
    //Going through the entire palette

    for (int x = 0; x < palsize; x++) {
        rgbvalues[x][0] = (*test).rgbRed;
        rgbvalues[x][1] = (*test).rgbGreen;
        rgbvalues[x][2] = (*test).rgbBlue;
        //Natural Order
        sortvalue[x] =
            256 * 256 * rgbvalues[x][0] + 256 * rgbvalues[x][1] +
            rgbvalues[x][0];
        //Luminance
        sortvalue2[x] =
            .299 * rgbvalues[x][0] + .589 * rgbvalues[x][1] +
            .114 * rgbvalues[x][0];

        if (x > 0) {
            if (sortvalue[x] < sortvalue[x - 1]
                && sortvalue2[x] < sortvalue2[x - 1]) {
                //If the sort values are not in order
                return 1;
            }
        }
        test++;
    }
    //If this is reached the palette is perfect
    return 0;
}

//
//Palette Sort
//Purpose:
//Sorts the palette
//Order: "Natural Order"
//
RGBQUAD *palettesort(RGBQUAD * pal, int palsize)
{
    RGBQUAD *test = pal;
    int newpalsize = 0;
    bool valueexists = false;
    int rgbvalues[palsize][3];
    int sortvalue[palsize];
    cout << "creating palette\n";

    for (int x = 0; x < palsize; x++) {
        cout << (*test).rgbRed << "\t" << (*test).rgbGreen << "\t" << (*test).
            rgbBlue;

        for (int y = 0; y < newpalsize; y++) {
            //Remove Doubles
            if (rgbvalues[y][0] == (*test).rgbRed
                && rgbvalues[y][1] == (*test).rgbGreen
```

```
                && rgbvalues[y][2] == (*test).rgbBlue) {
                valueexists = true;
            }
        }
        //If not double then add
        if (valueexists == false) {
            rgbvalues[x][0] = (*test).rgbRed;
            rgbvalues[x][1] = (*test).rgbGreen;
            rgbvalues[x][2] = (*test).rgbBlue;
            sortvalue[x] =
                256 * 256 * rgbvalues[x][0] + 256 * rgbvalues[x][1] +
                rgbvalues[x][0];
            newpalsize++;
        }
        valueexists = false;
        test++;
    }
    //Re-sort the colors
    int tempvalues[4];
    RGBQUAD *newpal = new RGBQUAD[newpalsize];
    RGBQUAD *point = newpal;

    for (int sort = 0; sort < newpalsize; sort++)
        for (int x = newpalsize; x > 0; x--) {
            //If the values need swaping
            if (sortvalue[x] < sortvalue[x - 1]) {
                tempvalues[0] = rgbvalues[x][0];
                tempvalues[1] = rgbvalues[x][1];
                tempvalues[2] = rgbvalues[x][2];
                tempvalues[3] = sortvalue[x];
                rgbvalues[x][0] = rgbvalues[x - 1][0];
                rgbvalues[x][1] = rgbvalues[x - 1][1];
                rgbvalues[x][2] = rgbvalues[x - 1][2];
                sortvalue[x] = sortvalue[x - 1];
                rgbvalues[x - 1][0] = tempvalues[0];
                rgbvalues[x - 1][1] = tempvalues[1];
                rgbvalues[x - 1][2] = tempvalues[2];
                sortvalue[x - 1] = tempvalues[3];
            }
        }
    //Saving the Palette
    for (int sort = 0; sort < newpalsize; sort++) {
        (*point).rgbRed = rgbvalues[sort][0];
        (*point).rgbGreen = rgbvalues[sort][1];
        (*point).rgbBlue = rgbvalues[sort][2];
        point++;
    }
    return newpal;
}

//
//Create Palette
//Purpose:
//Creates a palette by finding unique colors in the image
//Sends results through the Palette Sorter
//
RGBQUAD *CreatePallete(CxImage image, DWORD * size)
```

```
{
    int width;
    int height;
    RGBQUAD test;
    RGBQUAD *newpal = new RGBQUAD[*size];
    RGBQUAD *point = newpal;
    bool valueexists = false;
    width = image.GetWidth();
    height = image.GetHeight();
    *size = 0;
//Separate the colors from the image
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            test = image.GetPixelColor(x, y);
            point = newpal;

            for (int z = 0; z < *size; z++) {
//If the color values are already indexed
                if ((*point).rgbRed == (test).rgbRed
                    && (*point).rgbGreen == (test).rgbGreen
                    && (*point).rgbBlue == (test).rgbBlue) {
                    valueexists = true;
                }
                point++;
            }
//Add the color to the list if not already listed
            if (valueexists == false) {
                (*point).rgbRed = (test).rgbRed;
                (*point).rgbGreen = (test).rgbGreen;
                (*point).rgbBlue = (test).rgbBlue;
                (*size)++;
            }
            valueexists = false;
        }
    }
    return palettesort(newpal, *size);
}

//
//Main
//Purpose:
//Contains most of the processing
//
int main(int argc, char *argv[])
{
    if (argc < 3) {
        fprintf(stderr, "Bosque Palette Checker - Console steg tester\n");
        fprintf(stderr, "usage: %s input-file output-file\n", argv[0]);
        fprintf(stderr, "example: %s image.gif image2.gif\n", argv[0]);
        fprintf(stderr, "Currently the program only tests out its \n"
                "detection method using two files so that \n"
                "the differences can be seen.\n\n"
                "The program attacks two methods:\n" "Palette sorting.\n");
        return 1;
    }

    int i;
```

```
char filein[256];
memset(filein, 0, 256);
strcpy(filein, argv[1]);
char *extin;
FindExtension(filein, &extin);

for (i = 0; extin[i]; i++)
    extin[i] = (char) tolower(extin[i]);

int typein = 0;
typein = imageType(extin);

if (typein == -1) {
    fprintf(stderr, "unknown extension for %s\n", argv[1]);
}

char fileout[256];
memset(fileout, 0, 256);
strcpy(fileout, argv[2]);
char *extout;
FindExtension(fileout, &extout);

for (i = 0; extout[i]; i++)
    extout[i] = (char) tolower(extout[i]);

int typeout = imageType(extout);

if (typeout == -1) {
    fprintf(stderr, "unknown extension for %s\n", argv[2]);
}

printf("Loading image %s\n", argv[1]);
CxImage image;

if (!image.Load(argv[1], typein)) {
    fprintf(stderr, "%s\n", image.GetLastError());
    fprintf(stderr, "error loading %s\n", argv[1]);
    return 1;
}

printf("%s is %d by %d\n", argv[1], image.GetWidth(), image.GetHeight());
//Pallete
DWORD psize1;
RGBQUAD *ppal1;
ppal1 = image.GetPalette();
psize1 = image.GetPaletteSize();

if (palettecheck(ppal1, psize1) == 1) {
    //If Palette not web size or smaller
    //Then Recreate palette
    if (psize1 > 216) {
        ppal1 = CreatePallete(image, &psize1);
    }
    else                    //Otherwise re sort and reduce
    {
        ppal1 = palettesort(ppal1, psize1);
    }
```

```
        image.SetPalette(ppal1, psize1);
    }

    if (!image.Save(argv[2], typeout)) {
        fprintf(stderr, "%s\n", image.GetLastError());
        fprintf(stderr, "error saving %s\n", argv[2]);
        return 1;
    }
    return EXIT_SUCCESS;
}
```