Steganography:
Encoding an Image Inside Another Image

Christopher Dixon, William Fink, Tracy Jackson

***Abstract* – Trying to take the concepts and ideas of steganography and execute them using a computer can be a tough idea. Using MIPS assembly as well as the MARS simulator, we attempt to make a program that will encode as well as decode an image inside of another image. This will not only simulate the ideas of steganography, but it will also expand on them further.**

I. Introduction

The purpose of this assignment is to take two separate images and encoding one image inside of the other. This is what the main idea of steganography is. Steganography is the idea of hiding some sort of secret, whether it be a message, or in our case a picture, behind something that people would not expect to be hiding something [1]. For our program, we want to be able to not only encode some sort of image but decode out that image as well. An effective encoding algorithm is only as good as its decoding algorithm. Without a good decoding algorithm, there is a chance that the image will look very altered from its original. The intention by doing such a program will be to take two files that a user will be able to give and then take those two images and parse through them. Parsing through these images will give the size values of the image as well as the overall max value of the color of the image. From there, using a special encoding algorithm, the image will be encoded inside the other. Reversing that process will enable for the image to be decoded. After all is said and done, the program will either write one file that has the two images in it or it will write the image that was pulled out from the hidden image. The goal for this project is to have these images be encoded and decoded with as little degradation as possible. We know that there will likely be some degradation in the images, but reducing this to be a small amount would make the program more practical. If the image is noticeably altered, then someone using this program to hide something might worry that someone might figure it out. Using an image viewer will allow us to look at the images side by side and compare to see the differences.

II. Background

Having a small background and understanding about how image processing works is a very key and vital part to making this program successful. Each image contains a series of bytes. These bytes represent the pixels that make up the image. Being able to store and alter these values is what the whole process of encoding and decoding is all about. The program has to be able to do this effectively.

The concept of steganography has been around for centuries and has only evolved over time. Originally people like the Greeks would use this technique in order to hide messages and send them across their empire knowing that their methods would be hard to crack. As time grew on and centuries passed, many other methods were developed in order to hide these messages. One such method was using invisible ink to write down messages that could only be uncovered when close to light or heat [2]. This is so important to understand as it gives a background and information about what used to be done in order to send secret messages compared to what can be done in the present day using computers.

III. Methodology

A. Basic Logic and File Parsing/Storing

From the beginning, the first thing that can be made was the program would only work using grayscale images. This is done because grayscale pixels contain values from 0-255. This would enable the parsing and the changing of the values to be much easier than working with RGB values as these would require more code and arithmetic. Since resources are already limited because of the limitations of MIPS and the MARS simulator, grayscale images was the only way to go. The intent of the program would be to read the file in as a string instead of binary. This means that we would have to do some string parsing in order to find and isolate each of the pixel values. Having these pixel values as a string would make the RGB values much harder to isolate and change, which is another reason why we went with the grayscale images. With that in mind, the file is stored using .asciiz in MIPS. The idea of parsing the file as a string we feel will be a good challenge versus using the binary encoding.

Before starting to run the program, the user will have to enter the files that they want to use directly into MARS. The reason for this is due to time restraints, not being able to implement a way for the user to input their own files. There are three files that they user can interact with, fileName, fileEncode, fileWriteName. Each of these files will need to be filled in with the correct file location for the program to run properly. If using a Linux environment, the files have to be located in the same directory as MARS and the program itself. Using a Windows environment would include having the use the double slash "\\" between each of the directory locations. As well as the files that are listed as global variables for the program include the buffer and the arrays that will hold the

entire file, or in our case the string. The buffer we created was a set size of 300,000. While we know that there is a very good chance that this will never be filled unless the file size was very large, we wanted to be sure that this would never be an issue and therefore hard coded this buffer size to be so large. Each image has a corresponding array which will be populated during the encoding and decoding functions. Storing these values in the array would be a challenge to make sure that every value is stored properly in order for the encoding and decoding process to be effective.

Upon start up for the program, giving the user the ability to decode or encode an image is something that brings interaction to the front. There will be a prompt, that was also initialized in the beginning of the program, that states that if the user wants to encode the images, they will have to type 'e' and if they want to decode the images, they will have to type 'd'. From there, there will be a jump that will direct the user to the exact part of the program that they are expecting it to be.

If the user selects 'e', then the first step would include that files that would have to be read and parsed such that the image and its values can be stored into an array. In order for this to work successfully, the same code that is being used to open up the first file will be used to open up the second file, only changing the names of the file locations as well as the array which everything is being stored. The way that it can be done is by simply copying the same code and changing function names slightly. This was the solution that we know was wasteful, but it was more practical when it came to trying to read in two separate files with two separate names. As the file gets read into the program, there needs to be a buffer which will hold the string of the entire file. From there, the parsing will begin.

The way each file is set up is the exact same. The first two lines contain unnecessary values that can be skipped over during the parsing process. Following these two lines, there are two very important values that have to be taken out as well as stored. These values are the length and width of the image that is being parsed. This is important as it gives the size of the loops that are going to be traversed in order to populate the array. Following this, there is a value that is the max pixel size. This value is the max number that would appear in the parsing and it represents a pixel in grayscale. After this, the parsing of the values can begin, and the numbers are then stored into memory using the arrays that were previously mentioned. As stated above, this is done for the exact same for both files and if the user selects the encode option then both files will be done back to back.

B. Encoding the Image

Now that everything has been placed into the arrays, it is time to start encoding the images. First, the arrays that the data is stored in will have to be restored into another register for ease of access. From there, the dimensions of the image that is being encoded and multiply it together in order to have the total number of iterations that should be occurring. For sake of ease, NumberArray contains the image being encoded and NumberArray2 contains the image being used for the encoding. The first word is taken out of both arrays and stored into temporary registers. From there the word from NumberArray is divided by 10, then its quotient is multiplied by 10 and stored back into NumberArray. The word from NumberArray2 is then divided by 28. Its quotient is then added to the value that was just stored into NumberArray. Since we are loading the values in word by word, the end of the function has us adding 4 to the address to get the new word. The loop will then iterate all the way as long as it still has values left in the NumberArray2. An example of the arithmetic used for encoding is below in which you will notice that there is not much a difference between the two values:

$$\frac{155}{10} = 15.5 \rightarrow 15 * 10 = 150$$

$$\frac{130}{28} = 4.64 \rightarrow 4 + 150 = 154$$

C. Decoding the Image

If the user decides they want to decode an image, meaning they entered 'd' when prompted in the beginning, the process of parsing the images into arrays of strings is skipped over. There is no need to encode the image first. We are assuming here that the image that the user is trying to decode already has gone through the encoding process. This means that the program will jump immediately to the decoding section of the program.

The array containing the image will have to be loaded into a new address that will be easily accessible. From there, similar to what happens in the encoding section, the values that make up the size of the image will have to be multiplied together to get the number of iterations that the loop will have to go through. From there it is a series of steps to find the new image. The word gets loaded into a new register and is subsequently divided by 10. The remainder of this would then have to be multiplied by 28 and then stored back into the array. The function will loop as long as there are still values in the image. Similar to what happens in the encoding function, the address of the array that is being parsed has 4 added to it in order to find the new word that will be loaded in the next iteration of the loop. The intent here is to try and replicate the same process of the encoding but this time in reverse. An example of this algorithm is shown below:

$$\frac{155}{10} = 15.5 \rightarrow 5 * 28 = 140$$

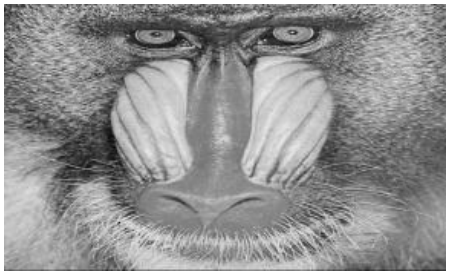
D. Storing and Writing the New Image

After either operation is done, the image will have to be restored into its perspective new file. In order for this to be done successfully, the image will have to be restored back into a string. This is done by taking the array, and after getting the word, the value that is stored there will have to be reversed. For example, this would mean that a value of 253 would then become 352. From there the value will be divided by 10 and then its remainder will be stored in ordered to form the string. So if the value is 253, then it becomes 352/10 which is 35.2 so 2 would be stored.

Finally after this has been done, the last step is to write to file. Based upon how the program is set up, the file that the user is saving the program to will already be stored inside the program. It will be found and then written using a syscall. It is important to add that they way that the program is currently set up, if the user wants to encode an image and then decode the same exact image, after the encoding process the user will have to go in a manually re-enter the file that they just encoded as the fileName variable in the program. This process, while very tedious, was done do to time constraints.

IV. Results

For testing purposes, we are using two images of the same size. These images are of an eye and a baboon. The eye image would serve as our image that gets encoded while the baboon is the message that is being used to encode the eye image. When decoding, the image that gets pulled out would be the baboon. These images are of size 289 and 174 and their gray scale values go up to 255. Having two images of the same size would give a chance to see the encoding and decoding process all the way through to see the outcomes. During testing, we concluded that having the buffer size so large, even with large files, never caused us to have any issues with the parsing ad storage. Both of these images are .pgm files as well as they make for the bitmap to be seen much easier. After the encoding and decoding processes are completed, the files will have to be opened in an image viewer to see the results. The Ifranview program is what we used to see the changes in the files. Below are the images that we used before any alterations were done. These are very important as they give an idea of what the images are supposed to look like. The screenshots were done using the Ifranview program which we mentioned above.



It is important to note that during the testing phase, one interesting aspect was realized. The instruction count for the program will vary for every iteration. This is because the program is so heavily depended on the images that are being used. If the images are small in size, then the program will not have to loop through for parsing and storing as many times as say a large image. In the case of the testing that was done, the baboon and eye files, being rather large, give an instruction count that is extremely high. If the user decides to decode the image instead of encoding two images, the instruction count would be significantly lower as the decode function skips over many other functions that are necessary for the encoding function to work properly, such as parsing the files. With this in mind, it is very hard to gage the instruction count for the program. During our testing, however, we found that our instruction count for encoding the images that were being used, was 9,425,132 and decoding the image was 5,939,761. Again, this is all theoretical and the instruction count would vary for each and every different file that the program is given to work with.

After these images were encoded, there was not much of a difference when opening the file in an image viewer. That way only slight change in the shade of gray but it was barely noticeable except in a few locations. This technique still proved to be very effective as to a naked eye, and to someone who had no idea there was an encoded image, it was be a complete shock. One of these reasons why this is because of the encoding technique did not alter the values of the pixels but not by much. This would allow for the image to do more or less retain its natural look without cause of suspicion.

As you can see, dismissing the differences in the text structure, the values are not much far apart. This is because the technique that was used for encoding has made it such that there is not much difference between the two images. The goal was to make the image look as natural as possible without making it look like it was altered too much. After this run had been concluded and the image was looked at in an image viewer, it was almost impossible to find many differences of the original images. We knew that there would be some degradation, but the differences were so slim it was very hard to notice, especially if you had no idea that something had changed with that image. Overall, this had proven that our theory and our method had worked. Below is the image of the eye after it was encoded with the image of the baboon:

As for the decoding process, we took the file that was just encoded and used that for the decoding process. We were able to compare the hidden image that was decoded to the original image. We noticed that there was a slight bit of degradation between them. Due to the encoding that was done on the image, we knew there would be a high possibility that we were off by at most, 14 pixel values for each byte. This was something that we noticed when opening the file up and seeing the individual byte values. Below, the first image is a sample set from the image the way it is supposed to look and the second is a sample set after the image was decoded:

```
113 64 97 70 72 84 99 82 81 145 101 69 73
150 116 142 114 115 90 99 149 138 133 132
```

```
112 56 84 56
56 84 84 56
56 141 84 56
56 84 168 112
84 141 112 141
112 112 84 84
```

As you can see, ignoring the way the strings as set up, there is only a slight difference between the way the image is supposed to look and the image after it was decoded. After further looking into this and looking at the image itself, there is definitely more degradation in the image when being decoded then there is in the image that is being encoded. We realized that this has to do with the way the formula is set up and trying to reverse the same process, but this time using the remainder from the division versus the quotient. However, with that being said, after looking at the images carefully using the image viewer, we noticed that there is not too much of an issue with the image after its been decoded and that the differences were still very slight even looking at the image only and not the pixel values. Overall, we would say that we achieved our goal for the decoding process as well. Below is the baboon that was decoded out of the encoded image:

V. Conclusion

Overall, this project has shown the abilities as well as the limitation of doing a program in MIPS. Having the ability to see each register is a great thing to have while only having a certain number of available registers can be detrimental to the design of the program. The hardest portion of this project was the parsing and the storing of the image. Having to parse through a set file and make sure that every single bit was accounted for was a challenge in itself. From there the encoding and decoding process was something that had to be unique and challenging enough that at first glance no one would be able to recognize it. The formula that we used for encoding and decoding, was something that we thought was very effective and helped us meet the goals that we had set for ourselves in the beginning of the project phase. We were very pleased to see that the formula that we had used for the encoding and decoding allowed for us to see the images without much degradation and allowed for the image to be hidden without anyone having any idea there was an image encoded inside of another image.

If further time was allotted, the first thing that would be attempted would be to try and shrink the size of the program and remove some of the copies of functions that were used mainly for the parsing of both files. This would not only make the program size smaller but would also make it much more efficient. There is also a possibility that fixing this would decrease the instruction count that we know is very high. It would also be interesting to try and do this same program but instead of doing it in grayscale, during it using RGB values. Encoding and decoding an image using an image proved to be a challenging concept but the solution that was development has shown that it can be done. We also think that in the future we would like to try and go back and change the formula slightly to see if there is a way to get even less degradation then we already have. Finally, we would like to be able to add more user interaction with the program. We would have liked to have had the user take their own files and input them on starting the program instead of having to change the hard coded file names listed at the beginning of the program. Regardless of the time that was allotted and the things we would like to do differently, steganography by encoding and image inside of another image, was a challenge but a challenge that we were able to beat.

References

[1] L. H. Newman, “Hacker Lexicon: What Is Steganography?,” Wired, 26-Jun-2017. [Online]. Available: https://www.wired.com/story/steganography-hacker-lexicon/. [Accessed: 28-Nov-2018]

[2] A. Siper, R. Farley, and C. Lombardo, “The Rise of Stenography,” May 2005.