INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES
DE MONTERREY

DEPARTAMENTO DE CIENCIAS E INGENIERÍA

Hiding Messages with Steganography

by

Raymundo Romero Arenas
&
Alexis Vázquez Martínez

Programming Languages Final Project

QUERÉTARO, QRO. MAY 2021

ABSTRACT

Steganography is the art of hiding images at plain sight. It particularly studies message hiding within a recipient, but it differs from cryptology because this last one is meant to cypher the message so it cannot be read without a key, while steganography just want to avoid prying eyes. This project has the purpose of showing different approaches of steganography practices to show how different programming paradigms behave on a common problem. During the chapters we will describe the problem presented, explain the different languages and the paradigms that they use, and finally show both solutions and the obtained results. Besides that, we will also present the architecture, the code and how to apply the solution in the real world.

Contents

Chapter 1

Context of the problem

1.1 Copyright

With all the information available on the internet at just a click away, it is getting harder and harder to keep track of who is really the owner of some media, an image for example. Many images use watermarks or texts that could be edited or simply cut from the original image, taking away credits from the original author.

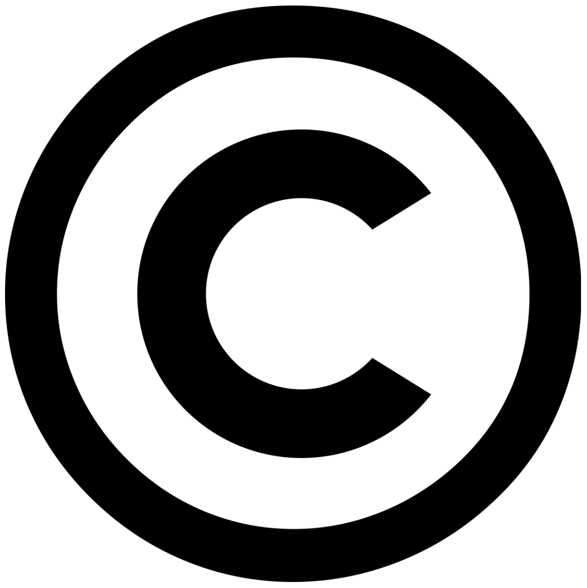


Figure 1.1: Copyright logo

According to the Copyright Law of the United States, the author of an image is the only one that can reproduce the copyrighted work, but this, as we know, is not always applied and sometimes it is extremely hard to succesfully obtain the origin of an image. The main problem usually is to artists, from digital art creators to photographers, which content is available of being public on the internet and the author is not even mentioned.

On 2017 an article went viral when an image of a monkey started to appear on social media, the problem was when different websites and media started publishing the image without the consent of it's author. The author told The Guardian that "If everyone gave me one pound for every time they used the picture, I would probably had 40 million pounds in my pocket, the income from that picture should have made confortable and I'm not". This happened because of two main reasons, the first one was that PETA sued the author because it claimed that the monkey was the author, not the photographer, and the second was that the images started to reproduce everywhere, so authenticity became hard to track.

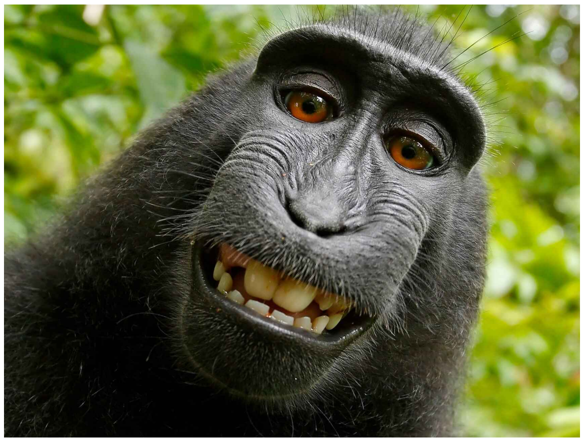


Figure 1.2: Naruto, the monkey of the picture

1.2 Why steganography?

Copyright is needed to give credits to the author, but keeping track of it is the hard part. Watermarks sometimes ruin the image and text on the sides usually ends being cut. With steganography, a "tag" on the image could avoid visible content on it but still serve as proof of ownership when requested. Also, with a parallel aproach, automaticed processes to create copyrighted images when uploaded would become extremely efficient and avoid computing time.

Chapter 2

State of the art

2.1 What is steganography?

Steganography is defined as the practice of hiding secret messages inside of *something* that is not secret. This *something* can be anything, such as images, documents, code or even audio[1].

The main purpose of steganography is to conceil and deceive. It can be done with any medium to hide messages, but they can still be found. However, it is not a form of cryptology because it doesn't use a key. While cryptography is a practice that enables privacy, steganography is a practice that enables secrecy.

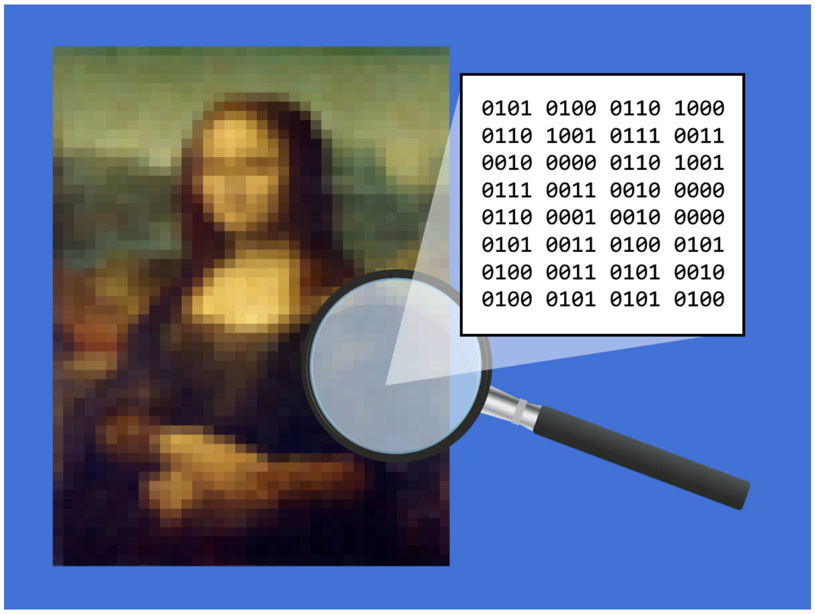


Figure 2.1: Hidden message example

2.2 History of steganography

The history of steganography dates back to Heródoto in 484 B., where in his book ”Las Historias” narrates how someone hid a message in a plank covered with wax, while others tattooed it in the bald spot of their slaves, allowed them to let his hair grow and the sent them to the recipient with instructions to shave their head. During the tow world wars, there were techniques such as invisible ink, microdotting, navajo code and null cipher to hide military communications.

With the arrival of modern steganography in 1985, first attempts were based on image watermarks, masking and LSB coding. Today, it has been recently used in espionage and terrorism. For instance, in 2001 it was studied the possibility that the 9-11 terrorists coordinated through hidden messages on the Internet, while in 2010 the FBI revealed an investigation in which it implicated ten russian spies of transmitting classified content from the United States government through digital steganography.

2.3 Steganalysis

Steganalysis is the main competitor to steganography, since it’s purpose is to attempt to defeat it [2]. What it tries to do is to find the hidden information and extract it or destroy it. There are many techniques and tools to do this:

2.3.1 Methods

Stego-only attack
We only have the stego-medium[1] and we want to extract the hidden message.

Known cover attack
We have both the stego-medium and the cover medium[2], so we can compare them.

Known message attack
We assume that we know the message and the stego-medium and we want to find the method used for hiding the message.

Chosen stego attack
We have the stego-medium and the steganography tool or algorithm.

Chosen message attack
A steganalyst generates a stego-medium from a message using one partticular tool.

[1]The final piece of information that the user can see.
[2]The medium in which we want to hide data.

2.3.2 Tools

There are some tools available that uses many steganography techniques to find hidden messages in different mediums. For example there is StegDetect, which focuses in image analysis. Another one could be Stego Suite, which is a more complete option that can also analize files, from documents to audio.

These tools show that the messages are not completely secure and the information can be easily obtained, that's why different steganography techniques are being studied and implemented.

2.4 Steganography techniques

2.4.1 Image steganography

An image is a numerical matrix that represents a rectangular grid of **pixels**, which are composed of three bytes each that defines its color according to the RGB Model. The higher the image quality and resolution, the easier it is to hide and retrieve a message. However, we must not change its format to avoid altering or damaging the inserted information. There are several methods to perform image steganography:

1. LSB (Least Significant Bit) = Replaces the least significant bit of every pixel with one from the message. Although it is the easiest method to implement, it is also the easiest to detect as it inserts white noise through lack of correlation between pixels (giving outlier colors in the image).

2. RGB Based = An indication channel chooses a random pixel to insert the bits of the message depending of its values. Offers the same security as LSB, but more storage capabilities.

3. PVD (Pixel Value Differentiation) = Substitutes the difference value between two pixel blocks with the bits of the message. It takes advantage of the human sight sensitivity to grey scales to make it more secure.

4. PMM (Pixel Mapping Method) = Defines a pixel seed that decides the pixels to use depending on its intensity value. Then it checks if the chosen ones or its neighbors are within the image limits on not, and inserts the bits of the message through a mapping of two or four message bits per neighbor pixel.

2.4.2 Audio steganography

Audio is a phenomenon in which a set of molecules vibrate through a sound wave that travels through a natural medium (air, water, land, etc...) an reaches to our ears. With an analog-to-digital conversion, we can graph any moment of the sound wave (called **sample**). Each sample has a channel, rate and size. When the sample of various channels are grouped in a time instance, we generate a **frame**, which is calculated by multiplying the number of channels by the sample size in bytes, which varies depending of the sampling rate. Thus, a frame in 5.1 surround sound (which has six channels) of four bytes length will have 24 bytes in total. There are several methods to perform audio steganography:

1. LSB (Least Significant Bit) = Replaces the least significant bit of every audio frame with one from the message to hide. Although it is the easiest method to implement, it is also the easiest to detect as it inserts white noise in form of a whistle that can be heard by the human ear. However, the audio quality doesn't diminishes as it has a theoretical probability of fifty percent of not changing the target bits.

2. Parity coding = Breaks the signal into regions and inserts a bit from the message in a parity bit. If both bits don't match, the process inverts the LSB in one of the region samples to give the user more than one option to hide them.

3. Phase coding = Replaces the phase of an audio segment with a reference one representing the secret information to hide. The remaining segments are adjusted to preserve the relative phase between them.

4. Speed Spectrum = Spreads the secret infomration throughout the audio frequency spectrum using an independent code. As a result, the signal uses a larger bandwidth than the required one to transmit the message.

5. Echo hiding = Divides the signal in blocks, inserts every bit from the message wherever it produces a hidden echo in it and concatenates the blocks at the end. It provides a higher data transmission speed and it is more robust than other techniques.

Chapter 3

Solution

3.1 Programming paradigm

3.1.1 Multi-Threads

When computers started to gain popularity, their processors were single threaded, depending only on one core. Soon, programs and tasks became a problem for traditional computing, since it depended on a single thread to execute, making it slow and costly.

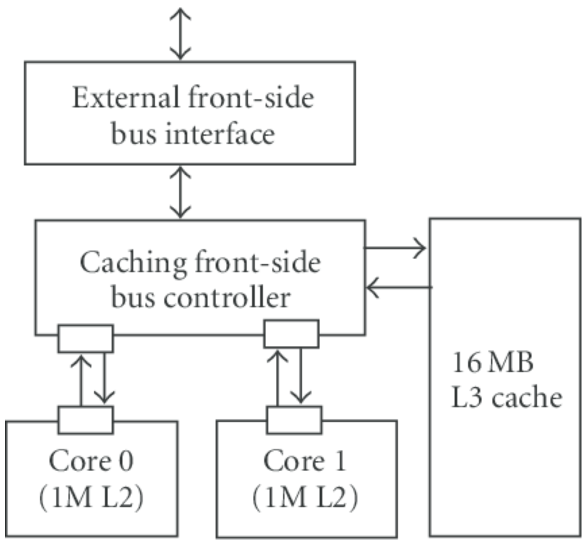

External front-side bus interface
Caching front-side bus controller
16 MB L3 cache
Core 0 (1M L2)
Core 1 (1M L2)

Figure 3.1: Multicore diagram

On 1986 a journal was presented and tried to explain what multiprocessing can do and how could it be benefical for every task that the computer wanted to perform[3]. Since then, parallel computing was relevant and widely applied, allowing software developers to use the architecture and take advantage of the multicore processors.

When executing a process, it involves different threads[1] to divide the work into subtasks and complete them faster since they divide the resources.

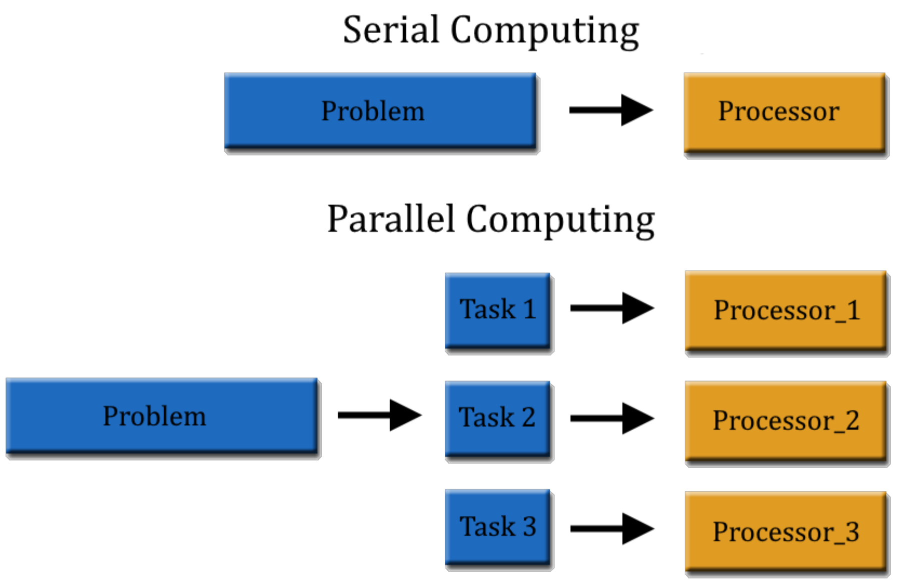


Figure 3.2: Parallel vs Serial computing

We are going to research both a parallel and serial approach to succesfully show how the execution times differ when the tasks are done sequentially and broke down into parts using threads.

3.2 Java

3.2.1 Java architecture

When Java was first presented, it offered a solution so that multiple operating systems could run the same program using it's own runtime environment. This allowed developers to use it without worrying about its technical implications. The idea is simple, the source code is compiled into Java using a runtime environment executable from any operating system. One huge advantage of Java is that allows parallel processing, allowing multi-thread operations to improve efficiency and speed of tasks execution.

[1]The smallest sequence of programmed instructions that can be managed independently by a scheduler.

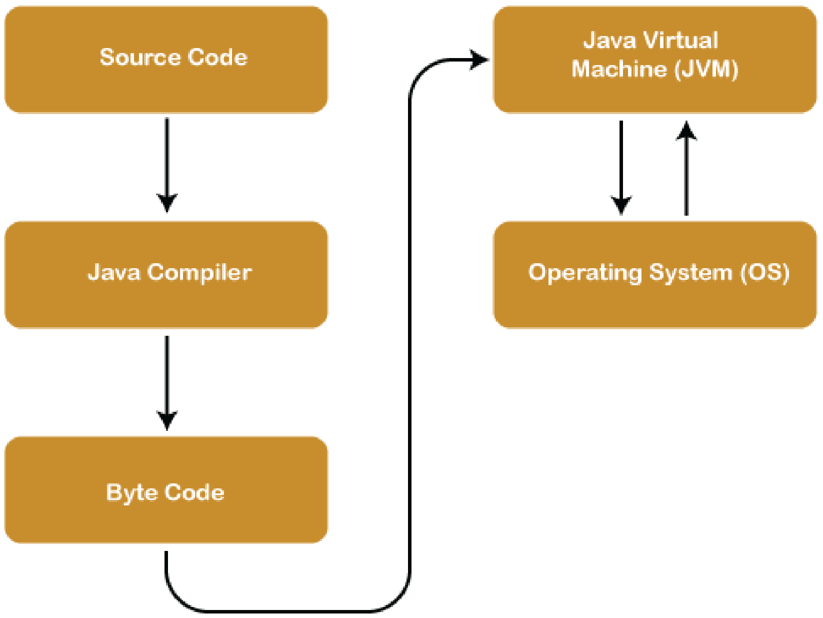


Figure 3.3: Java architecture diagram

3.2.2 Java implementation

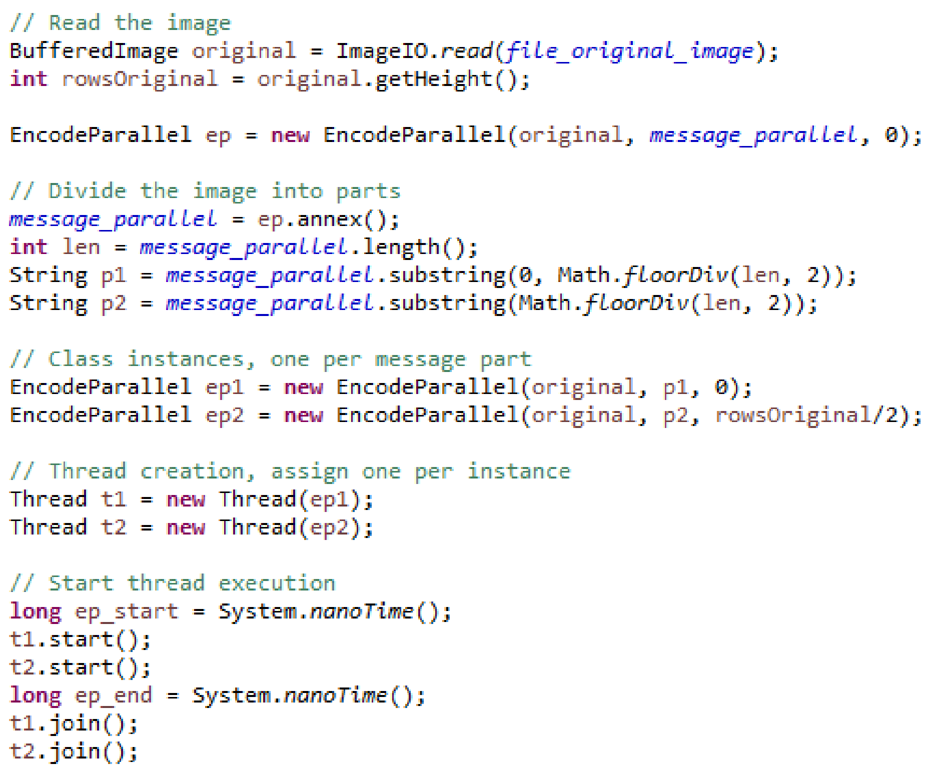
Our steganography implementation in Java starts with the definition of both approaches. First we defined how the hiding process would occur following a sequential and parallel programming paradigm, then we established the recovery method with the same idea on mind. When creating the GUI[2] we faced a new challenge using our method, but it will be explained in the next chapters.

Parallel

For the parallel implementation, we decided to divide the message into two different parts, and start two different threads to allow simultaneous execution.

First, the user provides an image and message to hide. We add a semicolon to the message to establish the end of it (we will cover this on the recovery process). Then, the message is divided in half by creating two different substrings. After that, we create two class instances, one per substring, and one thread per instance. Both threads are executed and we track their execution times. The first thread will hide the first half of the message at the beginning of the image, while the other one will do the second half at the middle of the image.

[2]Graphic User Interface



```java
// Read the image
BufferedImage original = ImageIO.read(file_original_image);
int rowsOriginal = original.getHeight();

EncodeParallel ep = new EncodeParallel(original, message_parallel, 0);

// Divide the image into parts
message_parallel = ep.annex();
int len = message_parallel.length();
String p1 = message_parallel.substring(0, Math.floorDiv(len, 2));
String p2 = message_parallel.substring(Math.floorDiv(len, 2));

// Class instances, one per message part
EncodeParallel ep1 = new EncodeParallel(original, p1, 0);
EncodeParallel ep2 = new EncodeParallel(original, p2, rowsOriginal/2);

// Thread creation, assign one per instance
Thread t1 = new Thread(ep1);
Thread t2 = new Thread(ep2);

// Start thread execution
long ep_start = System.nanoTime();
t1.start();
t2.start();
long ep_end = System.nanoTime();
t1.join();
t2.join();
```

Figure 3.4: Parallel hiding

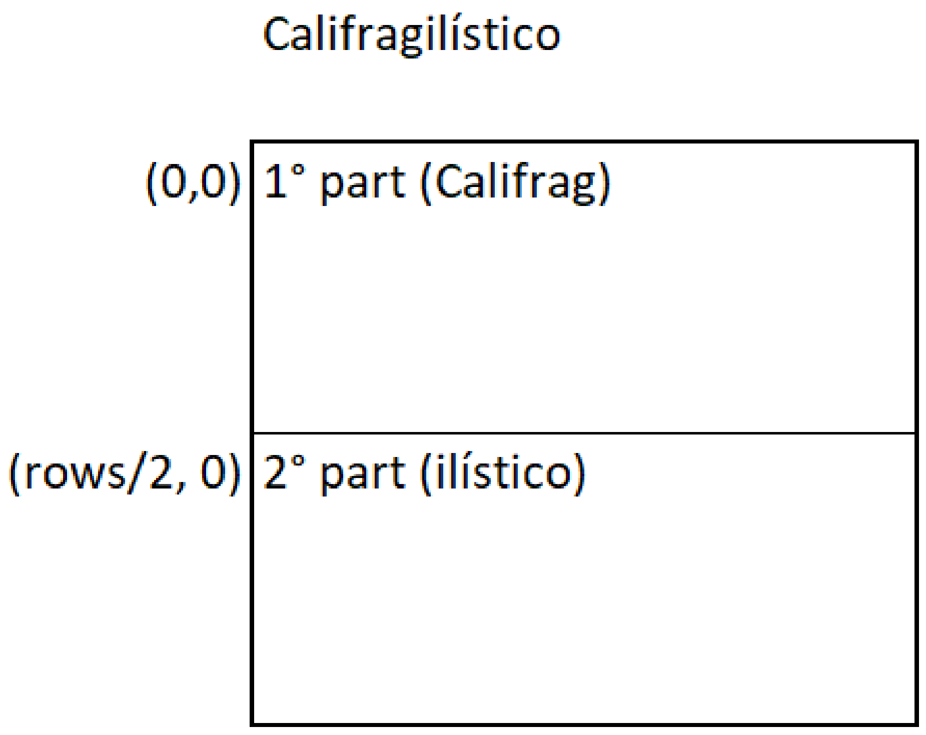
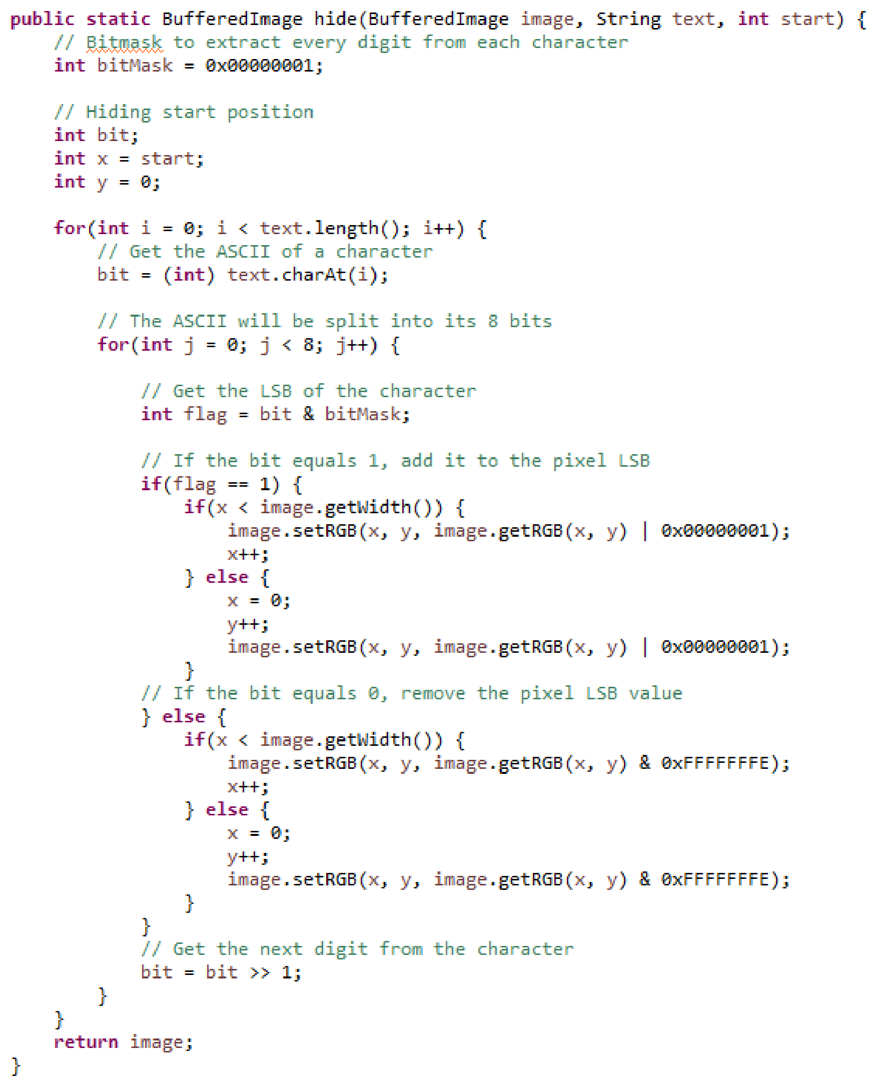


Figure 3.5: Parallel inside working

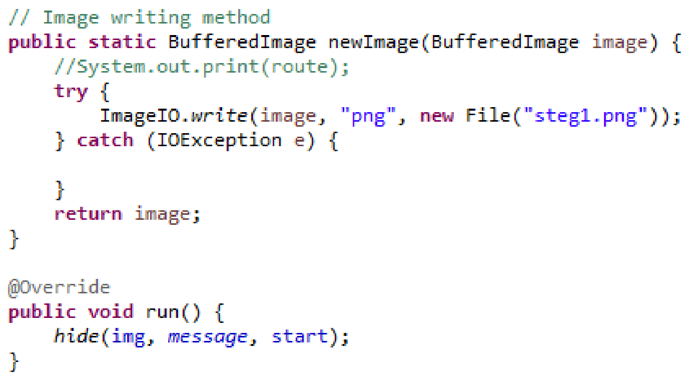


```
public static BufferedImage hide(BufferedImage image, String text, int start) {
    // Bitmask to extract every digit from each character
    int bitMask = 0x00000001;

    // Hiding start position
    int bit;
    int x = start;
    int y = 0;

    for(int i = 0; i < text.length(); i++) {
        // Get the ASCII of a character
        bit = (int) text.charAt(i);

        // The ASCII will be split into its 8 bits
        for(int j = 0; j < 8; j++) {

            // Get the LSB of the character
            int flag = bit & bitMask;

            // If the bit equals 1, add it to the pixel LSB
            if(flag == 1) {
                if(x < image.getWidth()) {
                    image.setRGB(x, y, image.getRGB(x, y) | 0x00000001);
                    x++;
                } else {
                    x = 0;
                    y++;
                    image.setRGB(x, y, image.getRGB(x, y) | 0x00000001);
                }
            // If the bit equals 0, remove the pixel LSB value
            } else {
                if(x < image.getWidth()) {
                    image.setRGB(x, y, image.getRGB(x, y) & 0xFFFFFFFE);
                    x++;
                } else {
                    x = 0;
                    y++;
                    image.setRGB(x, y, image.getRGB(x, y) & 0xFFFFFFFE);
                }
            }
            // Get the next digit from the character
            bit = bit >> 1;
        }
    }
    return image;
}
```

Figure 3.6: Steganography hiding process

The steganography process consists of turning the text into bits and insert them into the original image pixels.

Our first step is to transform a single character into its ASCII equivalent. Then, we split it into its 8 bits and use a bitmask of 0x00000001 with an ANDing operation to extract each one of them. Depending of the result of the ANDing, we insert the bit into the pixel. If the bit equals 1, then we use a bitmask of 0x00000001 with an ORing operation to insert it into the pixel, while if the bit equals 0, then we use a bitmask of 0xFFFFFFFE with an ANDing operation to nulify the LSB from the pixel. After we finish with a character's bits, we move on the next one until we traversed the whole image. In case we have used all the pixels from a row, we proceed on to the next one. Once we completed the steganography process, we write a new image with the modified pixels.

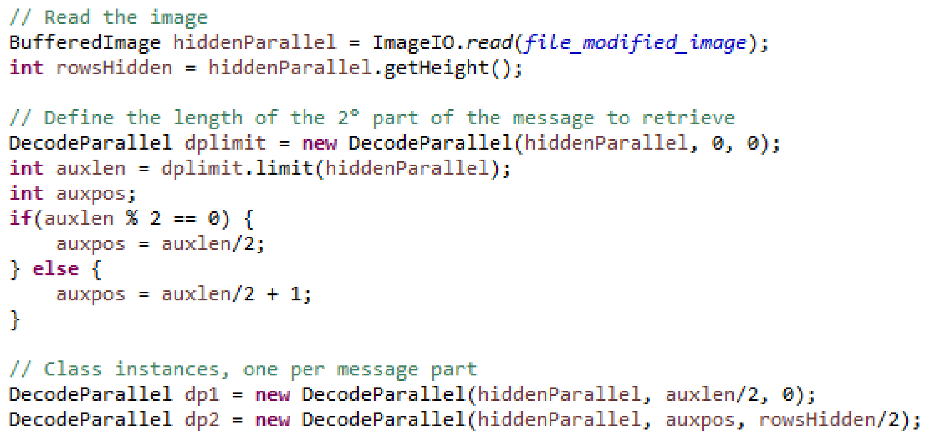


```java
// Image writing method
public static BufferedImage newImage(BufferedImage image) {
    //System.out.print(route);
    try {
        ImageIO.write(image, "png", new File("steg1.png"));
    } catch (IOException e) {

    }
    return image;
}

@Override
public void run() {
    hide(img, message, start);
}
```

Figure 3.7: Generate new image

Retrieving the message is when the process becomes tricky; our first attempt relied on knowing the length of the message, so it was difficult to tell the program when to stop sweeping the image without throwing garbage information to our output. That's was when we realized that a special character was needed to find the length of the message. As previously mentioned, we used a semicolon ";" to establish the end of the message and successfully determine the length needed to divide the message. The funcion *annex* in the Encode class adds the semicolon prior to the hiding process.

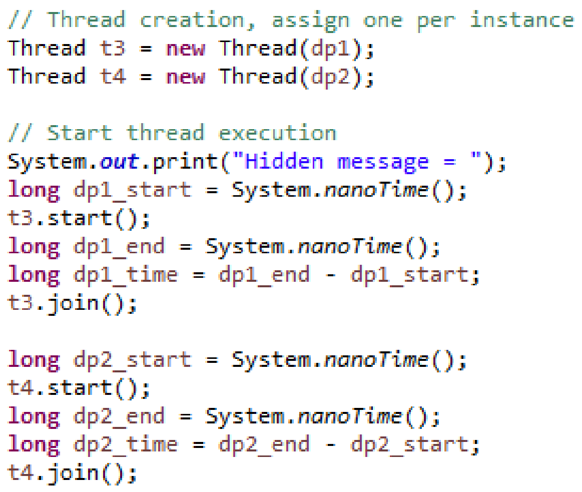
Before starting the recovery of the message, we need to first tell the threads how many characters they need to extract from their respective working areas on the image. We use a function called *limit* to find the semicolon we added during the hiding process to determine the length of the message. With this value calculated, we can specify to the threads the amount of characters each one of them will recover so we can concatenate them once they are finished.



```java
// Read the image
BufferedImage hiddenParallel = ImageIO.read(file_modified_image);
int rowsHidden = hiddenParallel.getHeight();

// Define the length of the 2° part of the message to retrieve
DecodeParallel dplimit = new DecodeParallel(hiddenParallel, 0, 0);
int auxlen = dplimit.limit(hiddenParallel);
int auxpos;
if(auxlen % 2 == 0) {
    auxpos = auxlen/2;
} else {
    auxpos = auxlen/2 + 1;
}

// Class instances, one per message part
DecodeParallel dp1 = new DecodeParallel(hiddenParallel, auxlen/2, 0);
DecodeParallel dp2 = new DecodeParallel(hiddenParallel, auxpos, rowsHidden/2);
```

Figure 3.8: Find the semicolon and assigns lengths to the threads

The recovery process works very similar to the hiding one: We use the bitmask 0x00000001 with an ANDing operation to extract the LSB of each pixel. Then we group them into 8 bits and transform them into an ASCII value and subsequently into its character equivalent.

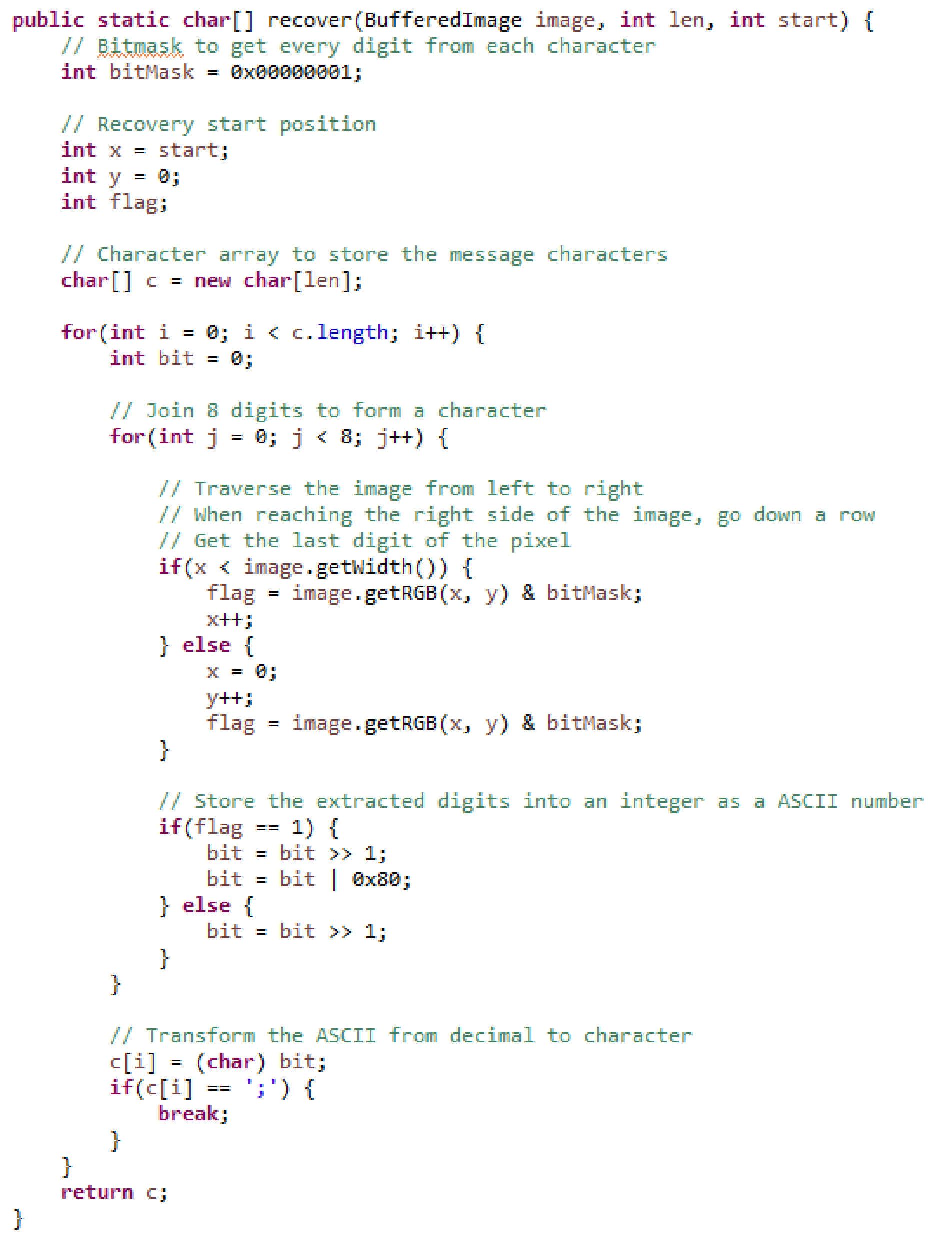


```
// Thread creation, assign one per instance
Thread t3 = new Thread(dp1);
Thread t4 = new Thread(dp2);

// Start thread execution
System.out.print("Hidden message = ");
long dp1_start = System.nanoTime();
t3.start();
long dp1_end = System.nanoTime();
long dp1_time = dp1_end - dp1_start;
t3.join();

long dp2_start = System.nanoTime();
t4.start();
long dp2_end = System.nanoTime();
long dp2_time = dp2_end - dp2_start;
t4.join();
```

Figure 3.9: Parallel recovery

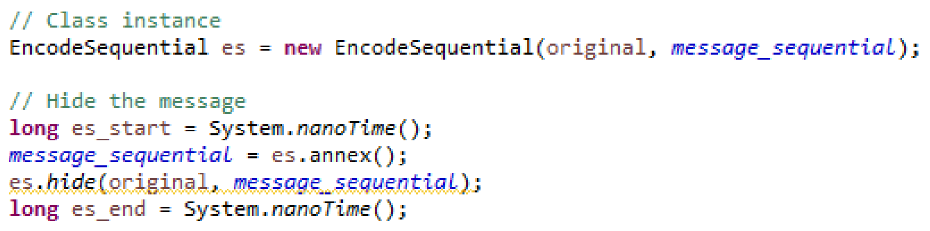
Sequential

For the sequential part of the code the idea is the same as the parallel one, same processes but single threaded. With a sequential approach, the necessary steps to do the steganography (such as the semicolon annex and message length calculation) were fewer. However, it is slower than parallel.



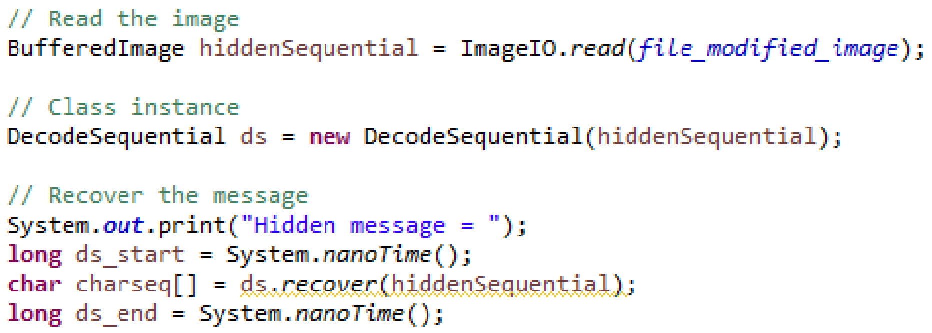
```java
public static char[] recover(BufferedImage image, int len, int start) {
    // Bitmask to get every digit from each character
    int bitMask = 0x00000001;

    // Recovery start position
    int x = start;
    int y = 0;
    int flag;

    // Character array to store the message characters
    char[] c = new char[len];

    for(int i = 0; i < c.length; i++) {
        int bit = 0;

        // Join 8 digits to form a character
        for(int j = 0; j < 8; j++) {

            // Traverse the image from left to right
            // When reaching the right side of the image, go down a row
            // Get the last digit of the pixel
            if(x < image.getWidth()) {
                flag = image.getRGB(x, y) & bitMask;
                x++;
            } else {
                x = 0;
                y++;
                flag = image.getRGB(x, y) & bitMask;
            }

            // Store the extracted digits into an integer as a ASCII number
            if(flag == 1) {
                bit = bit >> 1;
                bit = bit | 0x80;
            } else {
                bit = bit >> 1;
            }
        }

        // Transform the ASCII from decimal to character
        c[i] = (char) bit;
        if(c[i] == ';') {
            break;
        }
    }
    return c;
}
```

Figure 3.10: Steganography recovery process



```java
// Class instance
EncodeSequential es = new EncodeSequential(original, message_sequential);

// Hide the message
long es_start = System.nanoTime();
message_sequential = es.annex();
es.hide(original, message_sequential);
long es_end = System.nanoTime();
```

Figure 3.11: Sequential hiding



```java
// Read the image
BufferedImage hiddenSequential = ImageIO.read(file_modified_image);

// Class instance
DecodeSequential ds = new DecodeSequential(hiddenSequential);

// Recover the message
System.out.print("Hidden message = ");
long ds_start = System.nanoTime();
char charseq[] = ds.recover(hiddenSequential);
long ds_end = System.nanoTime();
```

Figure 3.12: Sequential recovery

Chapter 4

Results

Our main results proved that the execution time when using a parallel paradigm is more productive than the time obtained sequentially.

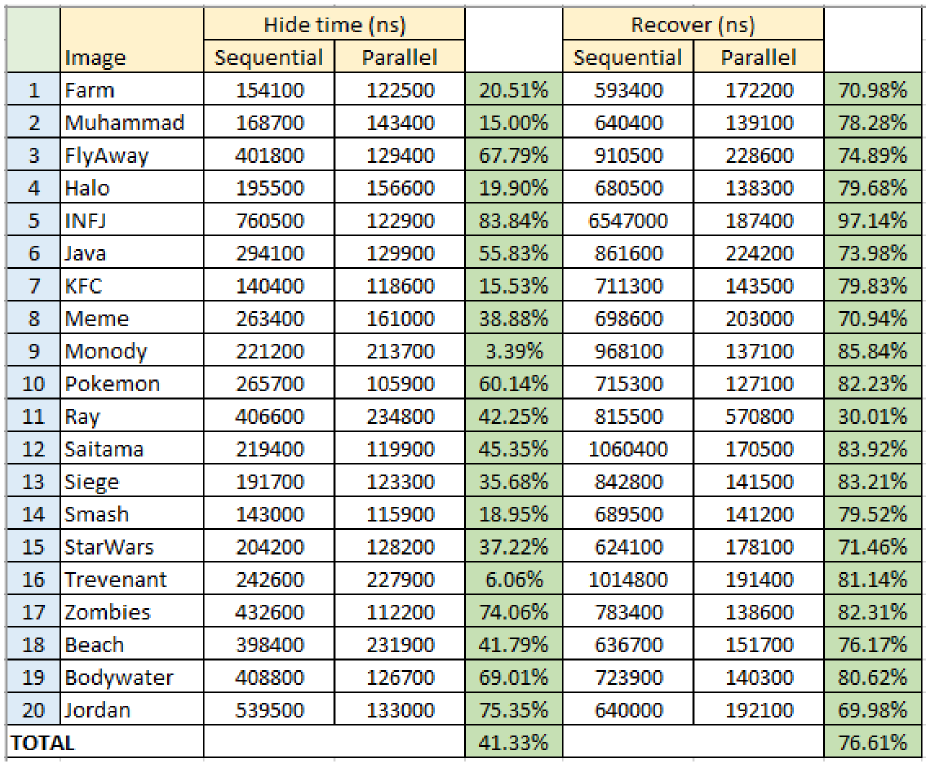


	Image	Hide time (ns)			Recover (ns)		
		Sequential	Parallel		Sequential	Parallel	
1	Farm	154100	122500	20.51%	593400	172200	70.98%
2	Muhammad	168700	143400	15.00%	640400	139100	78.28%
3	FlyAway	401800	129400	67.79%	910500	228600	74.89%
4	Halo	195500	156600	19.90%	680500	138300	79.68%
5	INFJ	760500	122900	83.84%	6547000	187400	97.14%
6	Java	294100	129900	55.83%	861600	224200	73.98%
7	KFC	140400	118600	15.53%	711300	143500	79.83%
8	Meme	263400	161000	38.88%	698600	203000	70.94%
9	Monody	221200	213700	3.39%	968100	137100	85.84%
10	Pokemon	265700	105900	60.14%	715300	127100	82.23%
11	Ray	406600	234800	42.25%	815500	570800	30.01%
12	Saitama	219400	119900	45.35%	1060400	170500	83.92%
13	Siege	191700	123300	35.68%	842800	141500	83.21%
14	Smash	143000	115900	18.95%	689500	141200	79.52%
15	StarWars	204200	128200	37.22%	624100	178100	71.46%
16	Trevenant	242600	227900	6.06%	1014800	191400	81.14%
17	Zombies	432600	112200	74.06%	783400	138600	82.31%
18	Beach	398400	231900	41.79%	636700	151700	76.17%
19	Bodywater	408800	126700	69.01%	723900	140300	80.62%
20	Jordan	539500	133000	75.35%	640000	192100	69.98%
TOTAL				41.33%			76.61%

Figure 4.1: Test results

4.1 Tests

As shown in the Figure 4.1, every single test obtained a better execution time when using a parallel paradigm. The time variation depends in external factors, but it still obtained in average a 41.33% of improvement when hiding the message and a 76.61% when retrieving it.

Chapter 5

Conclusions

20 tests were executed using images of different sizes and filetypes and the results were even better than the expected. As seen in the results, all tests showed that a parallel paradigm is more efficient than a sequential one, even if more steps are necessary to do the trick. When we divided the message into two using static threads, it used all the resources available, providing a faster execution time.

This tell us that using a parallel paradigm is beneficial to use successfully all the available resources in our computer. In our case, it showed that it improved the steganography process despite the message length and images sizes and file types.

Steganography using parallel programming became widely efficient and easy to implement, some of it perks are:

- Times reduced more than half, allowing quick usage.
- Better suited for modelling.
- Saved costs and improved resources

With the use of this program, users now can add their ”signatures” to the pictures and recover them easily without worrying of the available resources on their computers without worrying about using a particular OS[1] to execute it. As we saw earlier, there are many cases where copyrighting has a massive void. We expect that steganography can cover this issue in the foreseeable future for all kind of digital media.

[1]Operating system

Chapter 6

Usage

6.1 Repo

The repository of the code is located here, it can be accessed by everyone and it includes the instructions to execute it.

6.2 Instructions

How to run

This repo also includes a .jar file to run the project without compiling it in Java. The steganography program lets you hide and recover a message from an image and prints the parallel and sequential execution times. The jar file needs some information:

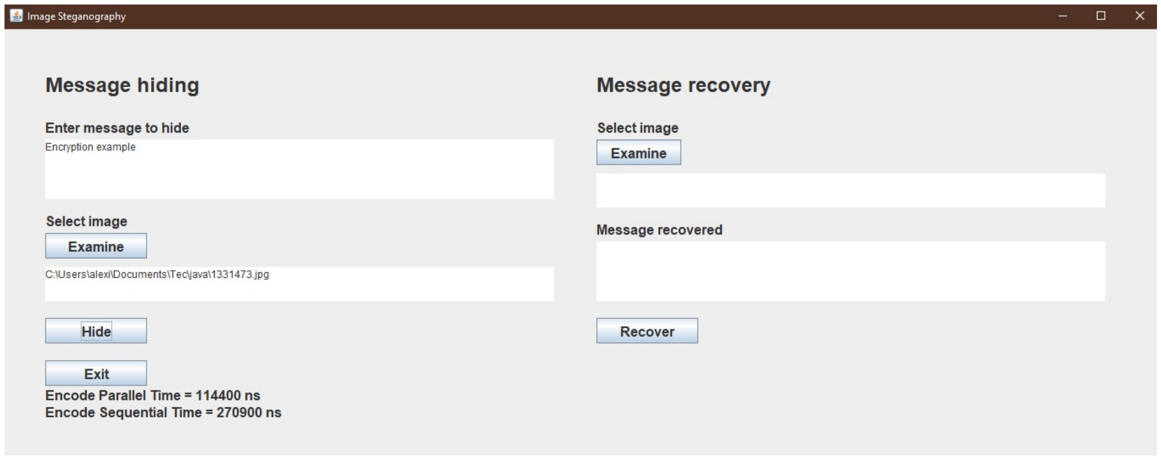
- A jpg image.
- A message to encrypt.

Hide message

Open the file ”stegjava.jar (or download it if it’s not). On the left side is the Hiding Menu, enter the message to hide, select the image to use and click ”Hide”. The result should be an image named ”steg1.jpg”.

To Decrypt

Open the file ”stegjava.jar (or download it if it’s not). On the right side is the Recovery Menu, insert the modified image and click ”Recover”. The result should be the recovered message.

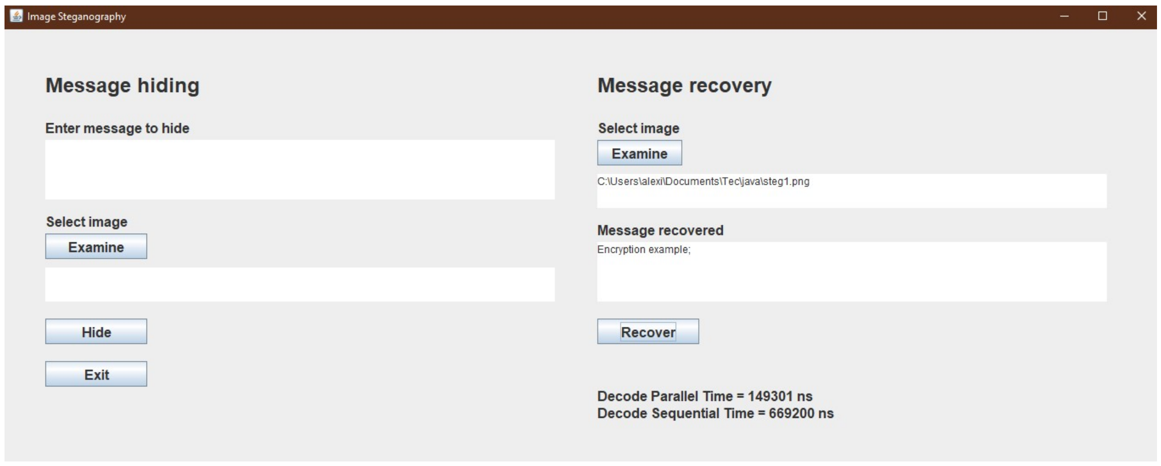


Encryption example

C:\Users\alexi\Documents\Tec\java\1331473.jpg

Encode Parallel Time = 114400 ns
Encode Sequential Time = 270900 ns

Figure 6.1: Encryption implementation in action



C:\Users\alexi\Documents\Tec\java\steg1.png

Encryption example;

Decode Parallel Time = 149301 ns
Decode Sequential Time = 669200 ns

Figure 6.2: Decryption implementation in action

Bibliography

[1] Stanger, James, The Ancient Practice of Steganography: What Is It, How Is It Used and Why Do Cybersecurity Pros Need to Understand It, *CompTIA*, Obtained from: https://www.comptia.org/blog/what-is-steganography

[2] Paladion, Steganalysis, *Paladion, High Speed Cyber Defense*, Obtained from: https://www.paladion.net/blogs/steganalysis

[3] Hicks H.R., Lynch V.E., An introduction to programming multiple-processor computers, *Science direct*, Obtained from: https://www.sciencedirect.com/science/article/pii/0021999186900884

[4] Kaspersky, ¿Qué es la esteganografía digital?, *Kaspersky*, Obtained from: https://latam.kaspersky.com/blog/digital-steganography/14859/

[5] Xataka, Cuando una imagen oculta más información de lo que parece: qué es y cómo funciona la esteganografía, *Xataka*, Obtained from: https://www.xataka.com/historia-tecnologica/cuando-una-imagen-oculta-mas-informacion-de-lo-que-parece-que-es-y-como-funciona-la-esteganografia

[6] Moreira et al., Análisis de técnicas de esteganografía aplicadas en archivos de audio e imagen, *Polo del Conocimiento*, Obtained from: https://polodelconocimiento.com/ojs/index.php/es/article/download/10/pdf

[7] IICybersecurity, ¿Cómo ocultar mensajes secretos en archivos de música?, *IICybersecurity*, Obtained from: https://www.iicybersecurity.com/audio-esteganografia.html