University Politehnica of Bucharest
Faculty of Automatic Control and Computers
Computer Science and Engineering Department

Diploma Thesis

Data Hiding Using Steganography

by

Monica Adriana Dăgădiță

Supervisor: Associate Prof. Dr. Eng. Emil Slușanschi

Bucharest, July 2011

Contents

List of Figures

List of Tables

Abstract

It is a truth universally acknowledged that "a picture is worth a thousand words". The emerge of digital media has taken this saying to a complete new level. By using steganography, one can hide not only 1,000 , but thousands of words even in an average-sized image. This art and science of data hiding, which has been used since ancient times, has become more and more popular.

This thesis presents various types of techniques used by modern digital steganography, as well as the implementation of the LSB (least significant bit) method. The main objective is to develop an application that uses LSB insertion in order to encode data in a cover image. Both a serial and a parallel version are proposed and an analysis of the performances is made using images ranging from 1.9 to 131 megapixels.

Chapter 1

Introduction

Throughout time, confidentiality has always been important. Whether it was carved in stone, written on paper or sent over the Internet, correspondence between two persons was exposed to tampering or eavesdropping. Therefore, it was necessary to provide a mechanism that protected written information.

The main objective of this thesis is to offer a solution for secure digital data transmissions. It will not be a networking solution that prevents data from being intercepted. More precisely, what will the application be trying to protect is the content of the data that is sent over the Internet.

One of the most common methods of securing transmitted data is cryptography (from Greek *kryptos*, "hidden, secret"; and *graphein*, "writing"). The purpose of cryptography is to disguise *plaintext* (data that can be read and understood without any special measures) in such a way that it becomes unreadable. The resulted text, called *cyphertext* can then be safely transmitted to destination. Knowing the algorithm used for encryption, only the receiver will be able to obtain the original message.

A very old, simple and famous method of encryption is *Caesar's cypher*, created by Julius Caesar himself in 50 B.C. It was created in order to prevent his secret messages from being read in case they fell in the wrong hands. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a shift of 3, A would be replaced by D, B would become E, and so on.

An important role in the evolution of cryptography was played by the Enigma Machine, an electro-mechanical machine used by Germans during World War II to communicate important military messages in a secure way. Before sending an encrypted message, the machine's electric and mechanical settings were set to a predefined combination (known only by the sender and the receiver of the message). The message was then written using the Enigma's keyboard and a

cyphertext was generated. The encoded text was then transmitted using Morse code. Having his Enigma machine setted to the same predefined combination, the receiver could easily obtain the original text.

After World War II, cryptography became more and more mathematical. The fast evolution of computers, the increased amount of data sent over the Internet, contributed to the development of modern cryptography. Data encryption became a must not only for military and government, but also for public institutions, organizations and common users. Widely used and known encryption algorithms are DES (Data Encryption Standard), AES (Advanced Encryption Standard), MD5, RSA and so on.

Access to powerful computers does not only mean better encryption. It can equally be a tool used for breaking a cipher or decrypting a message. No matter how powerful the encryption algorithm is, encrypted data has and always will arouse suspicion. This is where steganography comes in hand.

Steganography can be defined as "the art and science of communicating in a way which hides the existence of communication" [2]. The origins of the term are Greek, *stegos* meaning "cover" and *graphein* "writing". Although cryptography and steganography are often confounded, they are essentially different. Whilst the first one "scrambles a message so it cannot be understood", the other one "hides the message so it cannot be seen" [4]. It is commonly mistaken believed that steganography could replace cryptography. On the contrary, using the two techniques together one could create a solid and powerful encryption system.

Similarly to cryptography, steganography is an ancient technique. It is first mentioned by the Greek historian Herodotus in his work *Histories*, 440 B.C. He wrote about a noble man, Histiaeus, who shaved the heads of his most trusted slaves and tattooed messages onto their scalps. After the slave's hair grew back, he was dispatched with the message [1, 3, 6]. The method had its drawbacks - it could not be applied in case of an urgent message and the size of the text was obviously limited to the dimension of the slave's scalp.

Another ancient method of hiding data used wax tablets as a cover source. Greek people wrote text on the underlying wood and then covered it with a wax layer. Thus, the message was hidden under what appeared to be an unused tablet, that did not arouse any suspicion whatsoever.

In 1550, Girolamo Cardano, an Italian mathematician proposed a simple an ingenious method for hiding data. The secret information that both the sender and the receiver must be in possession of is a paper (or metal) mask with holes - the holes must be arbitrarily positioned and may come in different sizes. In order to hide a message, the mask is placed over a blank paper and the text is written inside the holes of the pattern. Afterwards, the remaining blanks must be filled in with a coherent text, so that the original message is completely disguised. An example of how this method works can be seen in Figure 1.1.

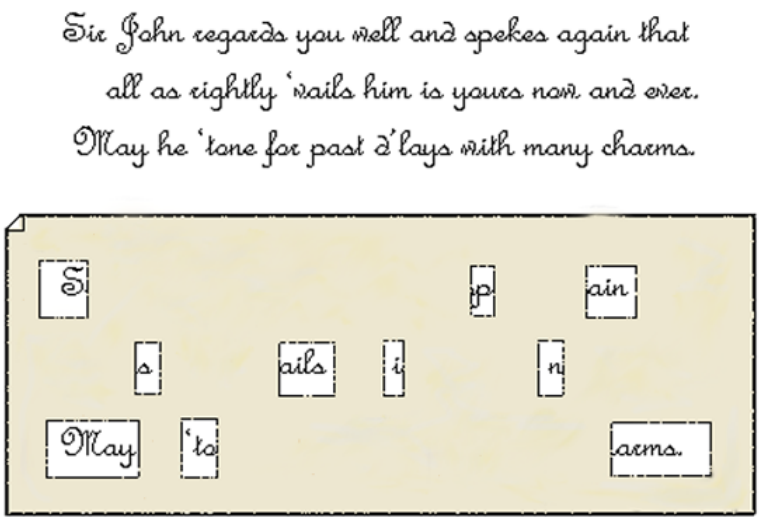

Figure 1.1: Example of Cardan Grille (wikipedia)

During and after World War II more and more techniques appeared : knitting Morse code messages onto the clothes of the couriers, hiding text on envelopes, under postage stamps, invisible ink - people used milk, vinegar or fruit juices to write secret messages. When heated, these types of fluids became darker and the hidden text could be read [6].

A commonly used tool by espionage agents was the microdot. A microdot is a text or image with size equal to a 1 mm disc. Due to the small dimensions, microdots could be easily hidden within a text or even on an envelope, without being noticed.

Later on, as information went digital, steganography changed. Messages could be hidden in the ones and zeros of text files, pictures, audio or video files. However, in order for the message to remain secret, only certain bits can be used. Hence, before the beginning of the information-hiding process, the stenographic system must identify the redundant bits - those bits that can be modified without altering the integrity of the file [5]. After this step, the least significant bits from the cover medium can be replaced with data that has to be hidden.

This thesis is organised into 8 chapters. In Chapter 2, a survey of digital steganography is presented. Chapter 3 briefly presents the hardware and software configuration of the system the application was developed and tested on. Next, in Chapter 4 the LSB insertion algorithm is described, along with the BMP file format used for the implementation. Chapter 5 discusses data structures created for the application, as well as the different ways in which it can be used - serial or parallel. Test cases and results obtained are analysed in Chapter 6, while the next two chapters present the conclusions the future work.

Chapter 2

State of the Art and Related Work

This chapter intends to offer a state of the art overview of the different types of steganography, and corresponding methods of steganalysis. Of all existent steganography techniques, digital image steganography will be the main subject.

2.1 Overview of Steganography

Almost all digital file formats can be used for steganography, but the most suitable ones are those with a high degree of redundancy. Redundancy could be defined as "the bits of an object that provide accuracy far greater than necessary for the object's use and display" [1]. According to this criteria, images and audio files are the most indicated.

However, information can be hidden inside any type of cover source, from simple text files to video ones. A very important feature of a cover file is the amount of data it can store without being changed noticeably, hence the redundancy requirement.

"Camouflaging" the message is essential. It is the main asset of steganography. Once the presence of a message is detected, its contents are no longer safe. This is why data encryption is recommended to be used in conjunction with data encoding. Instead of directly hiding a message, in a preprocessing stage it could encrypted and afterwards encoded into the cover file. For data recovery these steps would have to be executed in reverse - first data decoding and then decryption.

2.1.1 Text Steganography

This method is one of the most ancient ones. Some of the steganographic text procedures were described in the previous chapter. Another very famous approach is hiding the secret message in every n^{th} character of a text. It is not very secure, as knowledge of n automatically reveals the secret.

A safer way of data hiding is using a publicly available cover source, like a book or a newspaper and using a secret code that contains sequences of three numbers - page, line and character index. This way, the message can be revealed only by having both the secret code and the stego cover.

2.1.2 Image Steganography

Hiding information inside images is the most popular technique nowadays. The main reason is the high image traffic over the Internet, this type of media being the most frequently used. There are many methods of data encoding, almost one for every file format.

However different image-based algorithms are, there is one common factor: they all take advantage of the limitations of the human eye. If the message is hidden in such a way that the cover does not arouse suspicion, it will most probably not be discovered. On the contrary, if the noise in a image is visible or there are any other signs of the initial image being modified, a steganalysis tool will easily decode the hidden text.

2.1.3 Audio and Video Steganography

There are several ways in which information can be hidden inside an audio file. A common method is using the least significant bit. This way the modification of the initial file will not create audible sounds.

Another well known practice is "masking, which exploits the properties of the human ear to hide information unnoticeably. A faint, but audible, sound becomes inaudible in the presence of another louder, audible sound" [1]. Thus, a special channel is created.

Also based on human limitations, there are methods that use a person's inability to hear certain frequencies. It is possible to encode data using frequencies that are inaudible to the human ear. Above 20,000 Hz, no message will be detected [6].

Video files are generally collections of images and sounds. Consequently all methods applicable to these two file formats can also be used for videos. The main advantage is certainly the large amount of data that can be hidden. Having more space available, the message can be divided into smaller pieces and hid into different parts of the video file.

2.2 Image Steganography

As stated earlier, images are the most poplar files used as cover for data hiding. There are many different image file formats, most of them created for specific platforms or applications. For almost all image types there is at least one steganographic algorithm. All these algorithms could be grouped "into three categories, namely, spatial domain, frequency domain and adaptive methods" [3].

Besides these methods, there are a few that exploit the image format. One of them consists of writing the hidden message after the EOF tag of the cover file. The main drawback is that when opened with a text editor, the message will be visible to anyone, as shown in Figure 2.1.

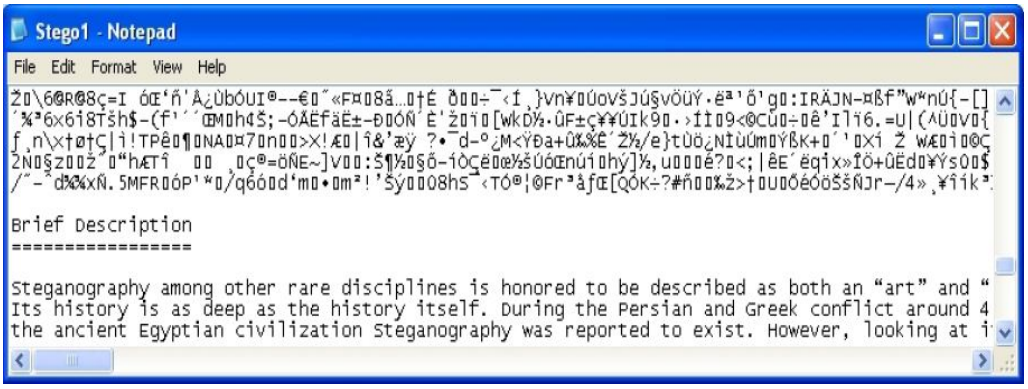

Figure 2.1: Stego-image opened using a text editor

Another easily detectable implementation of steganography is appending data into the cover image's EXIF (Extended File Information). EXIF is a standard used by digital camera manufacturers to store information like camera model, time picture was taken at, resolution, exposure time or focal length. The upside of this method is that there are absolutely no changes to the appearance of the image; the downside is that this information is easily viewable .

2.2.1 Image spatial domain steganography

Spatial domain methods are based on encoding at level of the LSBs. Least significant bit insertion is a common, simple approach to embedding information into a cover image. Depending on the size of the message and that of the image, one can use from the 1^{st} to the 4^{th} LSB. Using 4 bits however, is most likely to produce visible artefacts. In its simplest form, LSB makes use of BMP images, since they use lossy compression [1]. Pixels' values are stored explicitly, easing the process of data embedding.

Palette based images such as GIF files are also a very popular format, commonly used on the Internet. The format supports up to 8 bits per pixel, thus

allowing a single image to reference a palette of up to 256 distinct colors. GIF images are indexed images - the colors that appear in the image are stored in a palette, or lookup table. Each pixel's value is in fact the index of its color. One disadvantage of this method is that the new value of a pixel could point to a totally different color. If for BMP images modification of the LSB means a new but very similar color to the original one, in this case the new value is a new palette index. A solution could be sorting the lookup table so that similar colors have adjacent positions. In case there are less colors than the maximum possible number, a preprocessing of the image could add new colors which are very similar to the ones already existent.

2.2.2 Image frequency domain steganography

The most common file format used for this type of steganography is JPEG. It is a very popular file format, mainly because of the small sizes.

JPEG is a frequently used method of lossy compression for digital photography (image). The degree of compression can be adjusted, allowing a selectable tradeoff between storage size and image quality.

The first step of JPEG compression is converting RGB representation to YUV (Y corresponds to luminance or brightness, while U and V stand for chrominance or color). Taking advantage of the increased sensitivity of the human eye to changes in brightness, the color data is downsampled in order to reduce the total size of the file.

The second step is the actual transformation of the image; usually DCT (Discrete Cosine Transform) is used, but there is also possible to use DFT (Discrete Fourier Transform). The image is split into blocks of 8x8 pixels, which are transformed into 64 DCT coefficients each [1]. DCT and Fourier transform are very similar, in the sense that they both produce a kind of spatial frequency spectrum.

The next step is quantization. Yet "another biological property of the human eye is exploited : the human eye is fairly good at spotting small differences in brightness over a relatively large area, but not so good as to distinguish between different strengths in high frequency brightness" [1]. Consequently, magnitudes of the high-frequency components can be stored with a lower accuracy than the low-frequency components.

Last, the resulting data for all 8x8 blocks is further compressed with a lossless algorithm, a variant of Huffman encoding.

With a lossy compression scheme, JPEG files were at first considered useless for steganography. Hiding information into the redundant bits would have been of no use, since these are left out in the compression process. However, properties of this algorithm have been exploited in order to develop steganographic techniques. One of the most important aspects of JPEG compression is the fact that it is divided into lossy and lossless stages. DCT and quantization phases are lossy,

but the final stage, Huffman encoding is not. Consequently, LSB insertion could be done exaclty before Huffman encoding is applied. Encoding data at this stage is not only safe (as there will be no loss), but it is also difficult to detect since it is not in the visual domain.

2.2.3 Adaptive steganography

The previous two sections described algorithms that work either with the image spatial domain or the frequency one. There are however steganographic techniques that make use of both domains. These methods take statistical global features of an image before attempting to interact with its LSB/DCT coefficients [3].

Based on statistics, the most suitable places for data hiding will be identified. This way, the stego-message will be less likely to be discovered. As one of the simplest and most used methods of steganalysis is noise detectection, a good algorithm will avoid smooth areas of uniform color. Noisy images - wether noise is deliberately added or existent from the beginning - are the most suitable. A complex color scheme is also important, as variations will not be easy to detect.

Patchwork

Patchwork is a statistical technique that uses redundant pattern encoding to embed a message in an image [1]. The principle of the algorithm is the following: redundancy is added to the hidden information and then scattered throughout the image. By using a pseudorandom generator, two areas/patches of the image are selected. In one of them pixels are lightened while in the other they are darkened. This way average luminosity will not be affected.

One of the main disadvantages of this method is that only one bit is embedded. Consequently, the size of the stego-image should be chosen carefully. If the image is permissive enough, the secret message could be distributed into various places. This way, if one of the patches is destroyed, the others may still be intact.

2.3 Steganlysis

As more and more techniques of data hiding are developed, methods of detecting the use of steganography also advance [6]. Analogue to cryptanalysis, steganalysis is the science of detecting messages that were hidden using steganography.

In order to detect secret data, different image processing techniques can be applied: image filtering, rotation, cropping and translation . The stego-image's structure must be analysed in order to measure its statistical properties: histograms,correlations between pixels, distance and direction [3].

There are a few challenges steganography has to deal with:

- The suspect file may or may have not hidden data encoded into its contents.
- Hidden data can be encrypted before encoding, thus requiring a cryptanalysis method as well.
- The suspected file may contain noise of irrelevant data encoded, which can turn the steganlysis process into a very time consuming one.

If in cryptanalysis it is obvious that an intercepted encrypted data contains a message, steganalysis is not such an "exact science". The process usually begins with several suspect information streams, but nothing is for certain. The initial set of streams is then reduced to a smaller and highly probable to contain secret data one. It is not a very simple technique and positive results are never guaranteed.

As stated in the previous sections, cover data can be a text, an image and even an audio or video file. With the development of steganographic techniques adapted for each file format, different types of steganalysis methods have also emerged.

There are several methods of text analysis that detect certain patterns or disturbings, odd use of language or an unusual amount of white space [6].

Due to the increased popularity of digital image steganography, image steganalysis techniques are the most numerous ones. A widely used method involves statistical interpretation. While LSB might not seem very important, it actually can offer plenty of information regarding the content of an image. Figure 2.2 exemplifies the amount of data stored in the LSBs and the MSBs of an image.

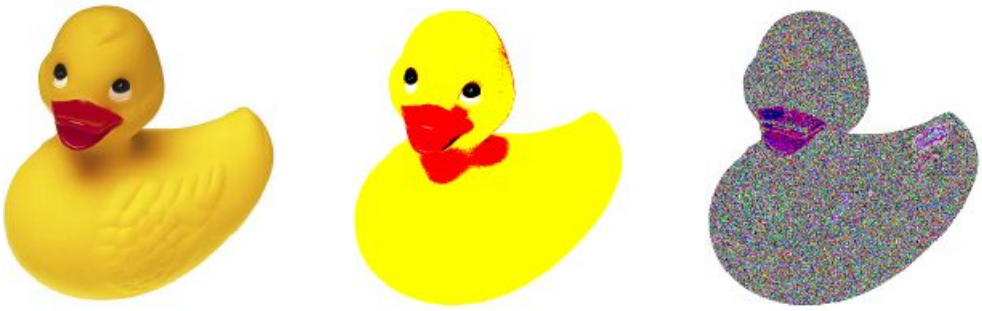

Figure 2.2: left:initial image, center:MSB sampling right: LSB sampling

The assumption that the least significant bits are mostly random, is wrong. Even the modification of a small percentage of LSBs could be easily detected. In fact, it is desirable that all the LSBs are modified, although the image's storage space is too large compared to the message's size. Figure 2.3 is an illustration of

this concept. Using only a slice of the stego-image could arouse more suspicion then using the entire image.

Figure 2.3: LSB planes after : left - no modification, center - modification of an amount less than the possible carrying capacity, right - padding of the image with random data past the significant data being hid

Statistical analysis may be also used against audio files. Added to this, there can also be detected high, inaudible frequencies that can be further scanned for information. Odd distortions or patterns in the sounds might as well point out the existence of a secret message. Other suspicion sources are differences in pitch, echo or background noise [6].

Chapter 3

Hardware and Software Configuration

All tests of the application were done on the NCIT Cluster of Politehnica University of Bucharest. For the profiling and development part I used the ibm-quad.q queue, and for testing the ibm-opteron.q one.

Table 3.1 contains hardware configurations for the LS22 and HS21 blades.

	LS22	**HS21**
Processor type	Six-Core AMD Opteron	Intel Xeon E5405
Sockets/Cores	2/12	2/8
Main Memory	16 GB	16 GB
Core Count	6	4
Core Speed	2,6 GHz	2 GHz
L2 Cache Size	512 KB	12 MB

Table 3.1: Hardware configuration of LS22 and HS21 blades

Both LS22 and HS21 blades are running Red Hat Enterprise Linux.
The tool used for profiling was Sun Studio Performance Analyzer.
The compiler used was gcc version 4.6.0.

Chapter 4

Algorithm description

Hiding information inside images is one of the most popular steganographic techniques used nowadays. The method I chose for data encryption is the LSB (Least Significant Bit) approach. The image format chosen for hiding data is BMP.

4.1 Bitmap Images

The BMP file format, also known as Bitmap, was introduced by Microsoft as a standard image format for Windows users, independent of the version of the operating system. The file format is now supported by multiple operating and file systems [2].

An important property of this type of format is the fact that all pixels are explicitly written in the file (a number for each component of a pixel : red, green, blue). One of the most commonly used type of bitmap files is that in which each pixel is represented using 24 bits - one byte (8 bits) for each color component. Using 8 bits for each color component gives 256 possibilities for each color plane, respectively 16,777,216 for one pixel. Altering the least significant bits will result in a color slightly different from the original one. What is most important to steganography is the fact that the human eye is unable to detect such differences.

A bitmap file can be divided into two main blocks: the header and the data. The header (54 bytes) contains information related to file size, data offset (offset to the pixel array), image width, image height and number of colors used. The bitmap data block is in fact the actual image, stored as pixels. An unique property is the fact that pixels are stored in reverse order - the first line of data corresponds to the bottom line of the image, and so on. Moreover, for each pixel the three color components are blue, green, red (not the usual RGB order).

4.2 Data Hiding Using LSB Insertion

Least significant bit insertion is one of the most commonly used methods of data hiding, due to its simplicity. The amount of data that can be hidden into an image depends on the size of the image and the number of least significant bits used for encryption. Having a 200 x 200 image and using only the least significant bit from each color component we get 120,000 bits that can be used in the process of data hiding. If we want to hide characters, the amount of data that can be encrypted is 15,000 characters - this means a text twice longer than the Declaration of Independence [2]. However, if we need more space we can always use a bigger image or more bits from each pixel, taking into consideration that the more bits we use, the lower the quality of the final image will be.

The process of data hiding is quite simple. Using one bit from each color component of a pixel, we get three bits per pixel; this means we need three pixels to hide a letter (one byte). Figure 4.1 contains an example of how letter $A(10000011)$ can be hidden into an image, using only the LSB for each pixel's color component. The first three lines are the original values for the three pixels needed (first column is red, second blue and the third green). The last bits from the next three lines hide letter A (underlined zeros and ones) - red color marks the bits that have changed in order to hide the data.

11011001	01000110	11100111
00010010	01100111	00001011
00111000	11010101	00100011
11011001	01000110	11100110
00010010	01100110	00001010
00111001	11010101	00100011

Figure 4.1: Example of LSB insertion

As shown in Figure 4.1, only some of the bits' values change. Actually, on average, LSB requires only half the bits in an image to change [4]. Moreover, even using both the least and the second least significant bit will not affect the image noticeably.

4.3 Data Recovery

The process of data recovery is a very simple one. The input needed is the image and the number of bits used. After reading the color values for each pixel, extract-

ing the corresponding bits and concatenating their values, the hidden character can be easily recovered.

An example of how data can be extracted from a stego-image is shown in Figure 4.2. In this scenario data was hidden using two bits from each color component. Extracting the last two bits of each number and grouping them eight by eight we obtain 01100001 (a), 01110000 (p), 01110000 (p), 01101100 (l), 01100101 (e). So, the hidden word is "apple".

11011001	01000110	11100100
00010001	01100101	00001011
10111000	11010100	00100001
11011011	01010100	11100100
00010001	00100110	01001011
01111000	11010101	00100010
11011001	01000101	10100111

Figure 4.2: Section of a stego-image

Chapter 5

Project implementation

This chapter discusses the two implementations of the algorithm described previously. Both encoding and decoding are implemented in a serial and parallel version (using pthreads). In order to balance the workload, the parallel algorithm is based on the boss-worker model, which will be treated in a future section of this chapter.

5.1 Data structures

Before implementing the algorithm I firstly created some data structures that would ease reading the bitmap image and processing it. Some of the most important ones will be described in this section.

In order to read the image header I defined two data structures : *header* (Figure 5.1) and *headerinfo*(Figure 5.2).

```
1 typedef struct {
2     unsigned short int type; /* identifier    */
3     unsigned int size;       /* file size     */
4     unsigned short int reserved1, reserved2;
5     unsigned int offset; /* offset to pixel array */
6 } header;
```

Figure 5.1: header data structure

The header of a BMP image is actually made of two sub-headers : the bitmap file header and the bitmap information header (DIB header). The *header* data structure contains information that is usually used to identify the file. A typical

application reads this block of data first to ensure that the file is actually a BMP and that it is not damaged. *HeaderInfo* data structure in Figure 5.2 offers more detailed information about the file : bits used for each pixel, image size, image dimensions - this data is generally used by applications in order to display the image.

```
typedef struct {
   unsigned int size;          /* header size */
   int width,height;           /* image dimensions */
   unsigned short int planes; /* colour planes */
   unsigned short int bits;    /* bits per pixel */
   unsigned int compression;   /* compression type */
   unsigned int imagesize;     /*image size in bytes*/
   int xresolution,yresolution;
   unsigned int ncolours;      /*number of colours */
   unsigned int importantcolours;
} headerInfo;
```

Figure 5.2: headerinfo data structure

To represent the whole image - headers and pixel array - I created *image* data structure (Figure 5.3) that contains the two headers and char matrices for each color component: red, green and blue.

```
typedef struct{
    header header;
    headerinfo infoheader;
    unsigned char **red;
    unsigned char **green;
    unsigned char **blue;
} image;
```

Figure 5.3: image data structure

5.2 Serial implementation

The three main functions of the serial implementation are *readBMP()*, *hideData()* and *getData()*. Each of them will be discussed in the following section.

5.2.1 Read Data

Reading the data from the input image is done pretty straightforward. The first step is to read the file header and save all the information stored there about the image. One important info contained by the header is the offset to the pixel array. Having this data, a *fseek()* to the beginning of the pixels' representation is enough. Afterwards, each pixel value is read and saved in an image data structure, as described in Figure 5.3.

Although the algorithm of data hiding does not depend on the pixels' color components order, it is taken into consideration that bitmap files have pixels stored in reverse - firstly the order of the pixels (beginning with the last line of the image) and secondly the color channels (first blue, then green and last red).

5.2.2 Hide Data

HideData() function receives two parameters: the name of the file that contains the data that needs to be encoded, and the address of an image data structure that contains the info collected with *readBMP()* function. The input file is an ASCII file, containing a list of files that must be encoded into the image (Figure 5.4).

```
1  books/20000_Lieues_Sous_Les_Mers.txt 914405
2  books/Abandoned.txt 422683
3  books/Alice.txt 148550
4  books/All_Around_the_Moon.txt 617609
5  books/Anna_Karenina.txt 1977476
6  books/Jane_Austen.txt 4435053
7  books/Monte_Cristo.txt 2612802
8  books/Shakespeare.txt 5572510
9  books/Three_Musketeers.txt 1296461
10 books/Wuthering_Heights.txt 661933
```

Figure 5.4: Example of input file

The reason I chose this format for the input file was to fully take advantage of the "hiding space" an image has to offer. In case the hidden message consists of more than one file, it would be easier to hide them all into a singe image (surely if its size allows to).

The main part of *HideData()* function is a loop in which each line from the input file is read and the corresponding file encoded into the image. In order to be able to reconstruct the initial files, the content of each file is preceded by a

special tag, specifying the name of the file. For example, before encoding the text in *Alice.txt*, the string *+books/Alice.txt++* will be hidden.

As previously described in chapter 4, in case only the least significant bit from each color component in used, we need three pixels to hide a letter. However, three pixels means 9 bits that can hide data. A letter is one byte, meaning 8 bits. Consequently, if a letter is hidden within every 3 pixels, lots of bits will be wasted.

In order to fully benefit from the amount of hiding space a image has to offer, the data is encoded into a contiguous way. This means the first letter of the message is hidden into the first two pixels and the last significant bits corresponding to the third pixel's red and green channels. The first bit of the second letter will be stored in the blue value of the third pixel, and so on. If there is more than one bit used for data hiding, the algorithm remains the same. The bits of the message are still hidden in adjoining pixels.

To implement this behaviour, I considered 3 indexes: *pixel_no*(this index is actually a composed one - one value for the column and the other for the line of the current pixel being modified),*color*(has values 0 for red, 1 for green and 2 for blue) and *bits_used* (memorizes the amount of data that has been hidden into the current channel's value of the pixel being modified). All these indexes are modified in a "round-robin" way, offering the exact location of the bit that is being hidden.

5.2.3 Recover Data

The process of recovering the data from an image consists of two parts : first, the input image is read and then the least significant bits from each color of each pixel is concatenated into the final message.

RecoverData() function is the inverse of *hideData()*. Depending on the number of bits used to cover data, the function builds up sequences of 8 bits from one, two or more pixels. After a complete sequence is obtained, the new character can be used - in order to reconstruct the tag before the contents of a file or simply its content. The main loop consists of two steps:

Step 1: Identify tag

Step 2: Recover file char by char

When an EOF character is encountered, the function loops back to step 1.

5.3 Parallel implementation

Before starting implementing a parallel version of the algorithm I made a profiling of the serial version. As shown in Figure 5.5, all three functions *readBMP(), hideData()* and *writeImage()* take almost the same time. The best way to reduce the amount of time was to overlap reading the input image and hiding the data

for the first part an then also overlap reading the stego-image and recovering data, for the second part.

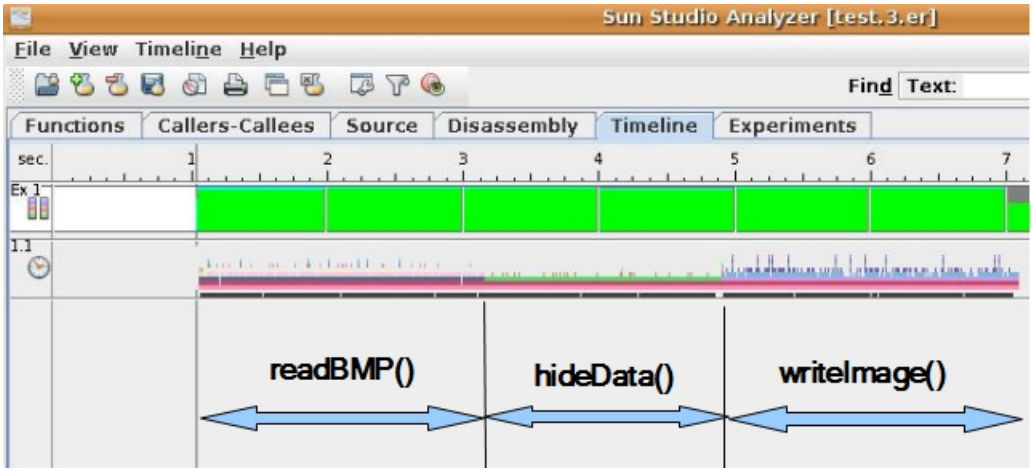

Figure 5.5: Profiling for serial data hide

In order to balance the work as much as possible I decided to use boss-worker model, which will be described in the following section.

5.3.1 Boss-worker model

Boss-worker model is very suitable for producer-consumer applications. One thread is designated as boss, while the other ones are workers. The boss thread obtains or produces jobs and places them into a queue. Worker threads then take jobs from the queue and process them.

Although generally there is only one boss, this is not a must. An application can have multiple boss threads. Moreover, any of the worker threads can behave as boss for a while - placing jobs into the queue; afterwards, it becomes a simple worker thread again.

One of the key factors of this model is synchronization. In order to to that the model usually uses a “thread-pool”. However, the thread pool alone cannot assure a correct behaviour - two threads can take the same job from the queue, in the same time, without knowing one about the other. To avoid this situation a mutex must be used.

A mutex (mutual exclusion) is an object that allows multiple threads to synchronise access to a shared resource. It has two states - locked and unlocked. Every time a thread needs the respective resource it must first acquire (lock) the mutex. If the resource is being used at that time, the thread will have to wait until the thread using the resource will free (unlock) the mutex. This way the

shared resource is never used by two threads (or more) simultaneously and data consistency is assured.

Another very important aspect of this model is the size of the queue. As it is not an infinite size resource, there may appear situations when it becomes full. In this case, the boss thread must wait until workers extract jobs from the queue, freeing the space for new ones. On the contrary, when the queue is empty the worker threads must wait until the boss thread places new jobs into the queue.

Every time a worker wants to extract a job from the queue, it must first obtain control over the mutex. If there is at least one available job, then it can process it and also send a signal to the boss thread (in case it was waiting for the queue to get empty). Alternatively, if the queue is empty the worker must wait until the boss thread creates a new job and places it into the queue.

Ending a Thread

In boss-worker model, each of the worker threads' function consists of an infinite loop in which a job is acquired and then processed. There is of course a moment when all jobs are done and worker threads must be somehow noticed that the queue will not be filled again. Making a thread stop when the queue is empty would be wrong, because the program could end before all jobs are even created. Therefore, there must be a specific way to announce the ending of a thread.

There are two possibilities to do this:

- After all jobs are created and placed into the queue, the boss thread creates one special exit-job for each of the threads and also adds them to the queue. This way, when a thread gets an exit task, it will end. However, this method works only if the queue's type is FIFO (first in first out) and jobs are treated in the same order they are created - exit-jobs must be the last ones in the queue.

- The queue can have a special "exit" flag. When the boss thread wants worker threads to end it can set this flag to a predefined value, associated with EXIT. Later on, if a worker thread wants to extract a job from the queue and the queue is empty and the exit flag is setted, the worker will break the infinite loop and end. Similar to the previous situation, the method does not always work - the essential condition is that both the flag and the status of the queue are checked before ending. Taking just the flag into consideration could lead to a premature ending of the threads, before all the jobs are processed.

After all worker threads have told to end, the boss thread waits for them to finish (join).

Balancing Issues

Balancing the workload is very important in order to obtain an efficient application. The key factor for this aspect is the queue. An efficacious application will have a queue that neither blocks the boss thread(threads), nor blocks the workers. In other words, the queue will never be completely full or empty.

To get to such an ideal situation, previous testing is necessary. The factors that can vary could be: the number of threads (both worker and boss), queue size, job sizes(if it is possible). Adjusting these factors should be quite easy. For example, if a boss threads blocks because the queue is full, then there are not enough workers (so it could be increased) or the queue is too small (thsi can also be easily fixed). Otherwise, if worker threads block because there are not enough jobs their number could be decreased or , alternatively, the number of boss threads could increase.

Nevertheless, if an application is controlled by external events (such as user or network input), balancing the queue will never be possible. In this case, the number ow boss and worker threads shoul be able to change dynamically. While the amount of work is relatively low, probably a single boss thread and a few workers are enough. In case the workload gets higher, the application should increase the number of workers (eventually boss too) by itself.

5.3.2 Algorithm implementation

Similar to the serial version of the algorithm, the parallel one also consists of two main parts : data hiding and data encoding. Both are implemented using the boss-worker method, described in the previous chapter.

The main functions are *boss()* - implements the behaviour of a boss thread, *worker()* - for the worker threads, *get_data()* - generates a new job to put in the queue and *process_tasks()* - initiates the queue , starts the boss thread and waits for it to finish.

Hide Data

The main advantage of the boss-worker model is that the data hiding process is divided into smaller jobs that are processed by the workers. The size of the job is very important - the total number of jobs should be at least equal to the number of threads. However, this would mean that the total processing time is setted by the slowest thread. On the other hand, having smaller jobs would allow faster threads to work more and considerably reduce the total amount of time.

In order to divide the workload, the input image must be split into small slices, one per job. The serial version of the algorithm depended of only one constant: the number of bits per pixel that can hide data. The parallel version depends on one more constant, the number of pixels per slice. Having the two

constants, the boss thread can compute the amount of data that can be hidden into a slice of the image. Figure 5.6 contains the exact structure of a job.

Slicing the image into small pieces also supposes dividing the text into fragments. The input file containing the data that must be hidden has the same structure as the one used for the serial version of the algorithm.

As shown in Figure 5.4, each line consists of a file name (path) and its size (in bytes). The boss thread will read this file and then, taking into consideration the number of bits used from each pixel and the size of an image chunk, it accordingly assigns chunks of files to each job. In order to ease the collection of the message for the decoding part, each job will have a tag. The tag contains the name of the file the message comes from and also the index of the message. Assuming the first file - *books/Alice.txt* - is divided into 3 parts, the tags corresponding to the three jobs would be: *books/Alice.txt0++*,*books/Alice.txt1++*,*books/Alice.txt2++*. The special mark *++* is used to mark the end of the tag. Before encoding the text from a job, the tag will be encoded. This way, when decoding the message, it will be written into a file that has the name of the tag. After all data is extracted, the initial files will be reconstructed using the files generated by the worker threads.

```
1  struct q_work_struct {
2    int req_type;        /* encode/decode */
3    FILE *fp;            /* image file pointerv */
4    int start_i,start_j;/* start pixel */
5    int end_i,end_j;     /* end pixel */
6    image **img;         /* image data structure */
7    FILE *in_file;       /* input file - to hide */
8    long num_bytes;      /* bytes to hide */
9    char *tag;           /* job tag */
10 };
```

Figure 5.6: Job data stucture

The roles of each field in the data structure presented above are the following:

- **req_type** - has value 0 for a hide-data-job and 1 for recover-data-job.
- **fp** - file pointer that points to the exact position where the worker thread must start reading pixel values.
- **start_i** - line of the starting pixel - it will be used by the worker threads to complete the pixel matrices from the image data structure.
- **start_j** - column of the first pixel, similar to the previous field described.

- **end_i** - line of the ending pixel.
- **end_j** - column of the last pixel read by the thread.
- **img** - pointer to an image data structure, used to store the stego-image (initial image + hidden message)
- **in_file** - pointer to the file containing the message that must be hidden (this field is only used for a hide-data-job).
- **num_bytes** - number of bytes that have to be hidden starting with the one that *in_file* points to (also used for encoding only).
- **tag** - tag of the file section associated with the current job (used only for hiding data).

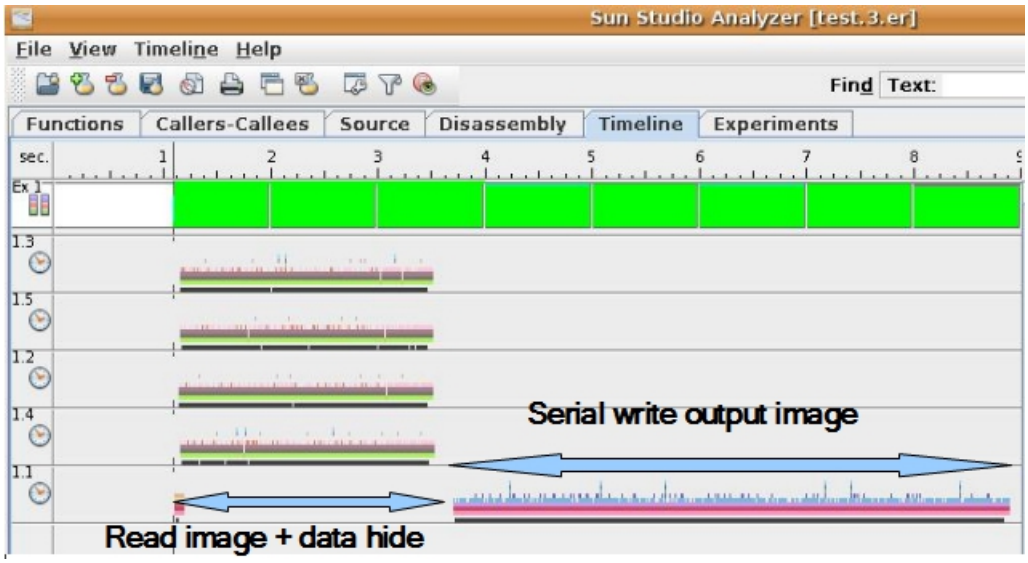

Figure 5.7: Profiling of parallel hide data - 4 threads

Figure 5.7 contains the profiling of the parallel version of data hiding, using 4 threads. There are two main sections: the parallel one, corresponding to image reading and data hiding and the serial one, representing *writeBMP()* function. Although the total amount of time decreased, there is still a problem - no matter how much the first section's duration will decrease, writing the final image will have always constant time, thus affecting the total speedup of the algorithm. The obvious solution would be for the worker threads to write the file concurrently. However, this would require a parallel file system. A compromise solution was to write parts of the output file separately and then concatenate them using a bash script.

Profiling of the second version of *hideData()* can be observed in Figure 5.8

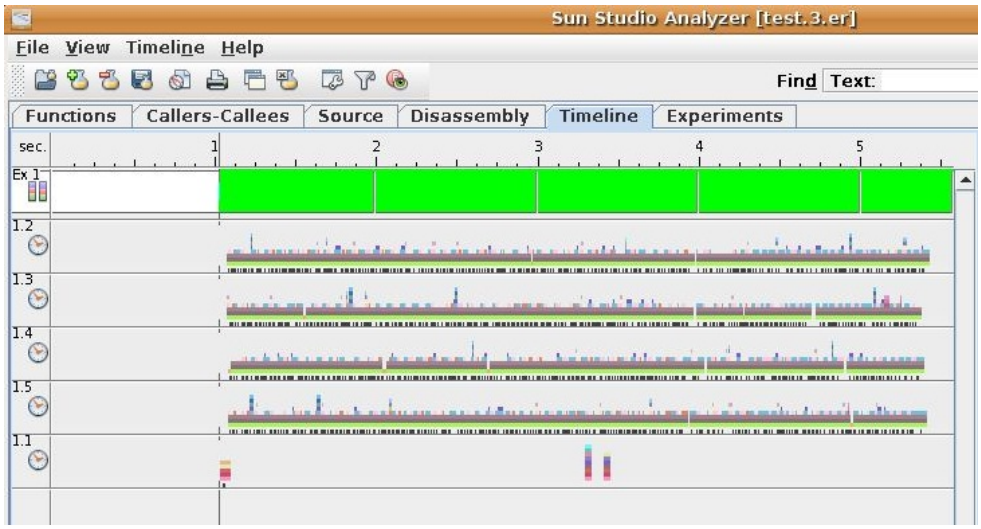

Figure 5.8: Profiling of parallel hide data + parallel write image - 4 threads

Recover Data

The process of recovering data is a simple one. After extracting a job from the queue, the worker thread starts reading from the pixel situated at (*start_i,start_j*). The message is then reconstructed char by char. As described in the previous section, every message begins with a tag. After the sequence *"++"* is encountered, a new file is created (the name is given by the tag). From this point on, each character decoded will be written into this file.

Profiling of the parallel version of data recovery is shown in Figure 5.9.

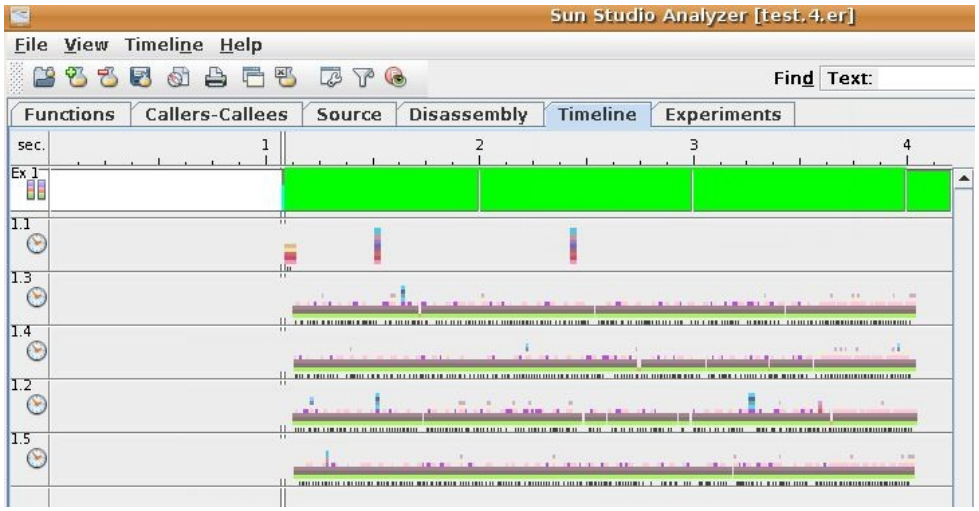

Figure 5.9: Profiling of parallel recover data - 4 threads

Chapter 6

Case studies

The first thing I tested after the serial version of the algorithm was complete, was the "capacity" of an image. The test image's size was 750 x 563. The input text files used for testing were books downloaded from www.gutenberg.org. However, capacity is not the only important thing. Maybe more important is the "ability" of an image to really hide the message it is carrying. In order to test both aspects - capacity and hiding efficiency - I used 1 to 8 bits to encode data. The results obtained are shown in Figures 6.1 and 6.2.

Figure 6.1: Output image after encoding - upper-left: 1 bit, upper-right: 2 bits, lower-left: 3 bits, lower-right: 4 bits

As shown in Figure 6.1, using up to four of the least significant bits in a color

component will not affect the quality of the image significantly. However, the last image, corresponding to the test where four bits were used, is visibly different from the first one. The amount of data hidden in this test case is 624 K of text, meaning the first nine and a half chapters of Jules Vernes' *Twenty Thousand Leagues Under the Sea*.

Figure 6.2 contains output images for tests in which 5, 6, 7 and even all 8 bits were used.

Figure 6.2: Output image after encoding - upper-left: 5 bits, upper-right: 6 bits, lower-left: 7 bits, lower-right: 8 bits

As expected, images are more and more noisier. The amount of data hidden is obviously higher, but any of the four images above would arouse suspicion. As mentioned before, the key factor in steganography is hiding the message transmitted. This is why there has to be a compromise between the size of the message transmitted and the aspect of the final stego-image. Luckily, nowadays image size is not a problem - a normal photo camera is now able to take 10 MP photos.

In order to test the scalability of the application I used 3 different image sizes:

Test case 1: size 1600 x 1200 (1.9 MP)

Test case 2: size 6048 x 4032 (24.3 MP)

Test case 3: size 13413 x 9821 (131.7 MP)

For each test case I ran the serial version of the algorithm and the two parallel versions (serial write and parallel write stego-image). In order to be able to hide a decent amount of data and not affect the image visibly, I used the two least significant bits from each color component of a pixel. The results obtained are presented in the following sections.

6.1 Test Case 1

This is the smallest image I ran tests on. With 1600 pixels width and 1200 height, the image can hide up to 1,440,000 bytes of data (using only the two least significant bits, as mentioned before).

6.1.1 Hide Data

The smallest time obtained for the serial version of the algorithm was **0.45 s**. Afterwards, tests were made using 2 up to 12 threads. Times and speedups corresponding to the first parallel version of data hiding are shown in Table 6.1.

Number of threads	Time (seconds)	Speedup	Efficiency
1	0.45	1	1
2	0.419	1.08	0.54
4	0.341	1.33	0.33
6	0.309	1.47	0.25
8	0.281	1.61	0.2
10	0.278	1.63	0.16
12	0.287	1.58	0.13

Table 6.1: Running times and speedups for parallel hide (version 1)

For this image size, serial writing of the output image did not affect the total time significantly. Still, I also tested the second version of parallel hiding data. As mentioned in the previous chapter, the output obtained are smaller files that must be concatenated in order to obtain the stego-image. However simple and straightforward the bash script used for this final operation is, the amount of time needed is about 0.2 s and it must be added to the total time needed for hiding data and creating the smaller files. Table 6.2 contains times and speedups for the second version of the parallel implementation.

Number of threads	Time (seconds)	Speedup	Efficiency
1	0.45	1	1
2	0.642	0.71	0.36
4	0.476	0.95	0.24
6	0.412	1.1	0.18
8	0.399	1.14	0.14
10	0.384	1.18	0.12
12	0.389	1.16	0.1

Table 6.2: Running times and speedups for parallel hide (version 2)

A better comparison between the performances of the two versions of parallelization is given by Figure 6.3.

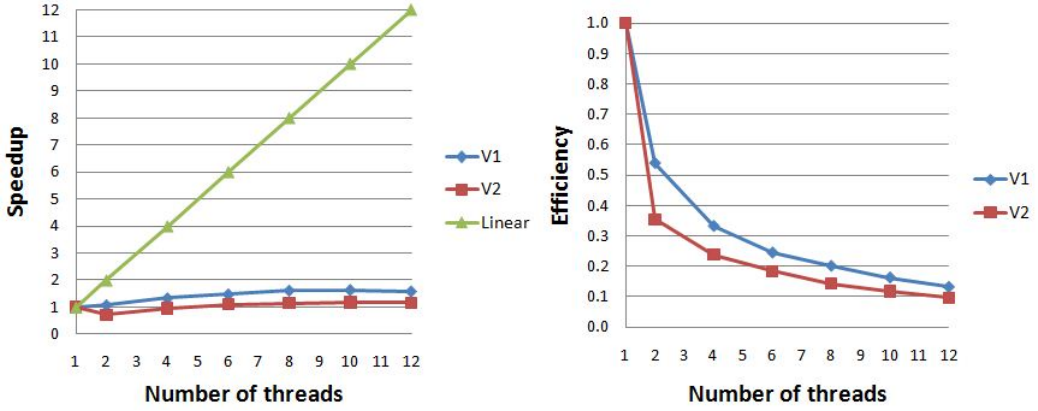

Figure 6.3: Hide Data - Version1 vs Version2 - left: speedup, right: efficiency

We can see that neither of the parallel versions of the algorithm brings a significant improvement in matters of time. Maxim speedup reached is 1.63, when running the first version of the parallel algorithm on 10 threads. Regardless the number of threads used, the first version seems to be more efficient. The main reason is the increased number of I/O operations compared to the amount of data processed, for the second version.

Figure 6.4: left: image before data hiding, right: image after data hiding

The input and output images can be observed in Figure 6.4. Using 2 bits from each color component to hide data, the final final stego-image contains 1.4 MB of text (Jules Vernes' *Twenty Thousand Leagues Under the Sea* and *Abandoned* and the first 9 chapters of Lewis Carroll's *Alice's Adventures in Wonderland*). There are no visible differences between the two images, meaning the message is safe to be transmitted, without risking to arouse suspicion.

6.1.2 Recover Data

This sections presents the results obtained for the second part of the algorithm, recovering the data hidden. Similar to data hiding, there are also three main steps: reading the input stego-image , decoding the message and writing it to an output file (or more).

Table 6.3 contains times and speedups obtained for this test case.

Number of threads	Time (seconds)	Speedup
1	0.24	1
2	0.55	0.44
4	0.45	0.54
6	0.43	0.57
8	0.43	0.57
10	0.44	0.55
12	0.44	0.55

Table 6.3: Running times and speedups for parallel recover

The small amount of data that can be hidden into a 1.9 MP image, makes the parallel version of data recovery inefficient. In all test cases - from 2 to 12 threads - the serial version proved to more adequate.

6.2 Test case 2

This section presents results obtained for the middle-sized image - 6048 x 4032 , meaning 24.3 MP. Because of the high resolution of the image, I decided to use 3 of the least significant bits in each color channel. This lead to a total of 27 MB of space available for the message, still without visible changes in the final stego-image.

6.2.1 Hide Data

For this test case, the total serial time for data hiding was **8.81 s**. The final stego-image contained 24 text files, amongst which: *Anna Karenina, Moby Dick, Les Miserables, The Count of Monte Cristo*, the complete works of Jane Austen and Jules Verne.

For the first parallel version of the algorithm, the smallest time was 3.013 seconds, for 12 threads. The slow decrease of the running times is a direct consequence firstly of the serial writing of the final stego-image (which takes about 2.7 seconds) and secondly of the intensive I/O operations.

As shown in Table 6.4, maximum speedup reached for parallel data hiding - version 1 - was 2.92. However, the most efficient test case was the one using 2

Number of threads	Time (seconds)	Speedup	Efficiency
1	8.81	1	1
2	4.920	1.79	0.9
4	3.776	2.33	0.58
6	3.496	2.52	0.42
8	3.214	2.74	0.34
10	3.152	2.8	0.28
12	3.013	2.92	0.24

Table 6.4: Running times and speedups for parallel hide (version 1)

threads - maximum value 0.9 . For more than 2 threads, even though speedup continues to slowly increase, the efficiency of the algorithm decreases fast. I did not use more than 12 threads firstly because it was the maximum number available and secondly because it would have not improved the total time.

Version 2 of parallel data hiding proved to be more efficient for this test case, as illustrated in Table 6.5. Concatenating the pieces of the output image using the bash script took 1.2 seconds. Added to this, the best time obtained was 2.28 seconds, again for 12 threads.

Number of threads	Time (seconds)	Speedup	Efficiency
1	8.81	1	1
2	6.01	1.46	0.73
4	3.92	2.25	0.56
6	2.83	3.11	0.52
8	2.64	3.34	0.42
10	2.41	3.66	0.37
12	2.28	3.86	0.32

Table 6.5: Running times and speedups for parallel hide (version 2)

The chart in Figure 6.5 contains the analysis of the two methods' performance. If version 1 was more efficient for the previous test case, for the 24 MP image parallel writing made a difference. For more then 6 threads, the speedup of the second version increases faster then the one corresponding to the first version of the algorithm. The evolution is not a spectacular one (and it could not be for this ratio of data processing versus I/O operations), but there is an improvement in matters of time.

Though 3 bits were used, the initial and the stego-image cannot be differentiated only by looking. A higher resolution means more details and therefore more places to hide data. However, using 4 bits would have been to much, because the output image would be too blurry. Figure 6.6 shows the 24 MP image before and

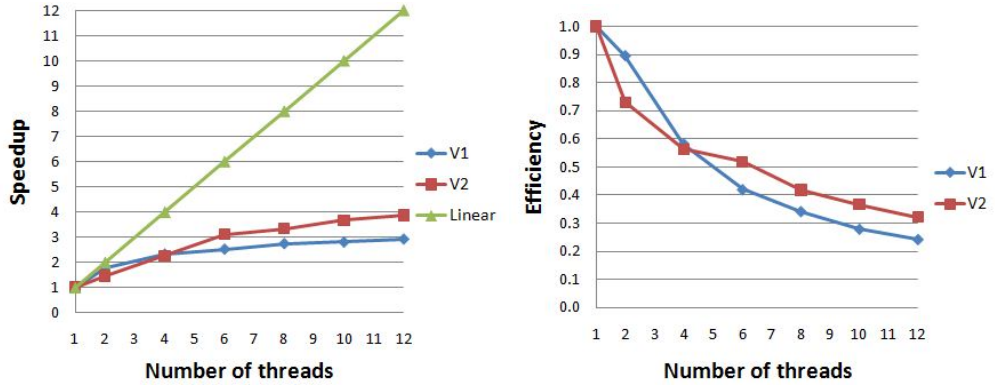

Figure 6.5: Hide Data - Version1 vs Version2 - left: speedup, right: efficiency

after data hiding.

Figure 6.6: left: image before data hiding, right: image after data hiding

6.2.2 Recover Data

Data recovery was also 'speeded up' almost 4 times with the parallel version. Once again the use of 12 threads brought the best performances, leading to a 3.92 speedup. The serial time was reduced from 5.12 seconds to 1.31.

Using 3 bits from each color channel and 500,000 pixel chunks, the image was divided into 49 slices, each containing around 0.5 MB of data. 49 slices equals 49 'recover-jobs' that were put in the queue by the boss. Each one of them was processed by a worker and the data recovered was written into a file, thus leading to 49 output files. All the output files were then unified using the python script in 0.72 seconds.

Table 6.6 contains times and speedups obtained for parallel data recovery for 2 to 12 threads.

Number of threads	Time (seconds)	Speedup
1	5.12	1
2	3.29	1.56
4	2.03	2.52
6	1.7	3.02
8	1.48	3.46
10	1.44	3.56
12	1.31	3.92

Table 6.6: Running times and speedups for parallel recover

6.3 Test case 3

The image used for this test case was the one with the highest resolution. It is a roadmap of Romania, having 13,413 pixels width and 9821 pixels height. I decided to also use 3 bits from each color component for this case. This lead to 148 MB of space to store the secret message.

6.3.1 Hide Data

With such a high resolution image and almost 150 MB of data to hide, the serial version lasted **47.88 s**. The first parallel version only managed to reduce the total amount of time to 15.05 seconds, for 8 threads. As Table 6.7 shows, maximum speedup reached was 3.18 for 8 and 12 threads.

Number of threads	Time (seconds)	Speedup	Efficiency
1	47.88	1	1
2	26.00	1.84	0.92
4	18.07	2.65	0.66
6	16.12	2.97	0.5
8	15.05	3.18	0.4
10	14.44	3.32	0.33
12	15.07	3.18	0.27

Table 6.7: Running times and speedups for parallel hide (version 1)

Similar to the first parallel version, the second one also reached maximum speedup for 8 threads. In this case performances were slightly better, with a maximum speedup of 3.85. The duration of the final image parts concatenation was 4.4 seconds, leading to a minimum time of 12.42 seconds. In terms of efficiency, the first version was better, with 0.92 for 2 threads. For the second version highest value was also obtained for 2 threads, but it was 0.89.

Number of threads	Time (seconds)	Speedup	Efficiency
1	47.88	1	1
2	26.85	1.78	0.89
4	16.55	2.89	0.72
6	13.84	3.46	0.58
8	12.42	3.85	0.48
10	13.93	3.44	0.34
12	15.99	2.99	0.25

Table 6.8: Running times and speedups for parallel hide (version 2)

The results in Table 6.7 and Table 6.8 can better be compared in Figure 6.7. Up to 4 threads both versions of the algorithm have the same efficiency. For 6 and 8 threads the second version's parallel write of the output file pieces and their concatenation is less time consuming and for more than 10 threads the overhead introduced by creating the worker threads is higher than the time saved having more workers.

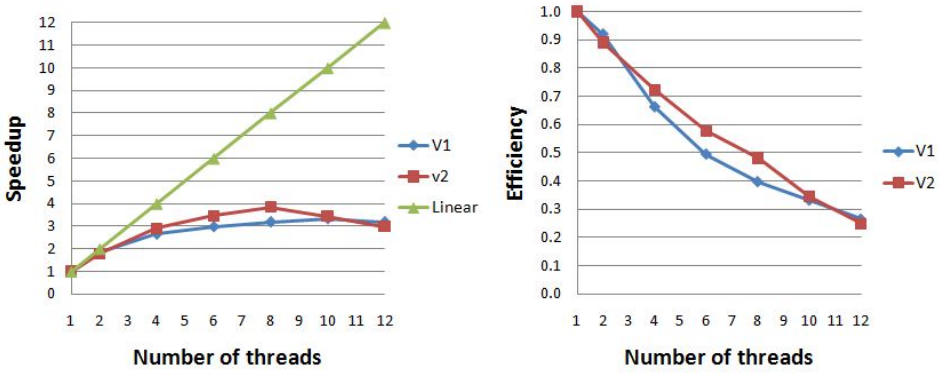

Figure 6.7: Hide Data - Version1 vs Version2 - left: speedup, right: efficiency

Figure 6.8 contains the initial image of the roadmap and the final image, after data hiding is done. Although the second image contains a 146 MB text message there are no noticeable differences.

6.3.2 Recover Data

Serial data recovery took **24.27** seconds. The parallel algorithm managed to reduce this time to a minimum value of 9.1 seconds, for 8 threads. With pixel chunks of size 2,000,000 , the image was divided into 66 slices. The resulting 66 output files created by the worker threads were merged into the final stego image in 2.94 seconds.

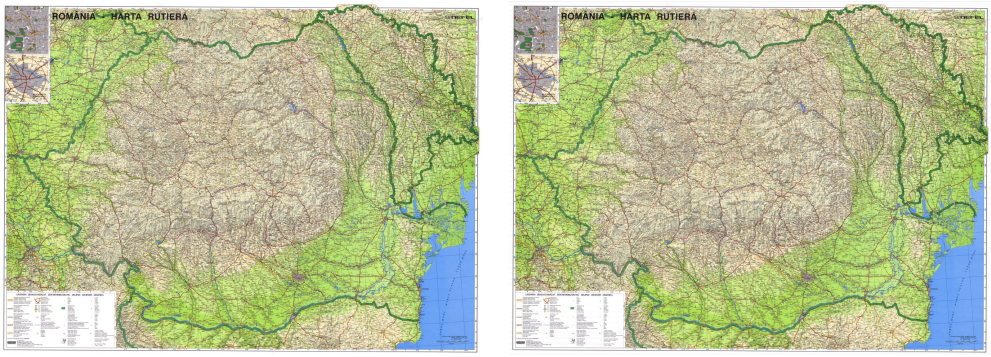

Figure 6.8: left: image before data hiding, right: image after data hiding

Times and speedups obtained for parallel data recovery using variable number of worker threads can be found in Table 6.9.

Number of threads	Time (seconds)	Speedup
1	24.27	1
2	15.23	1.59
4	13.56	1.79
6	11.21	2.17
8	9.1	2.67
10	10.18	2.39
12	12.21	1.99

Table 6.9: Running times and speedups for parallel recover

6.4 Image Hiding

The image size in the previous test case is obviously too large for sending just text. It is not often that someone wants to hide 130 MB of text. However, a 132 MP image could be used to hide another type of message - images. Text is not the only way two persons can communicate.

Lately image traffic over the Internet has increased exponentially. Why not use steganography to hide images? Whether the message is a text or an image, for the computer it is just a sequence of ones and zeros. So, if a text message can be hidden, so can an image.

The algorithm that I implemented takes as input ascii files. In order to hide images I used the *PPM (portable pixmap)* file format, which is very similar to the BMP format. The header however is very simplified - it contains an identifier of the PPM format, the width and height of the image and the maximum color value. A pixel is represented by three numbers, corresponding to its red, green

and blue components. In contrast to the BMP pixel matrix, in a PPM image pixels are stored in normal order, just as they appear on the screen. Figure 6.9 contains an example of a simple ppm image.

```
P3
# The P3 means colors are in ASCII, then 3 columns and 2 rows,
# then 255 for max color, then RGB triplets
3 2
255
255   0   0     0 255   0     0   0 255
255 255   0   255 255 255     0   0   0
```

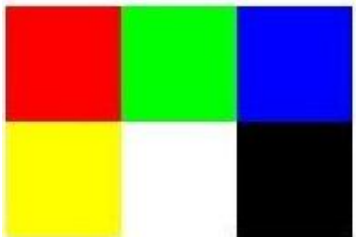

Figure 6.9: PPM file format and corresponding magnified image (wikipedia)

The file that I wanted to hide is an image of a galaxy, taken by one of NASA's space telescopes. Its dimensions are 4,000 pixels width and 4,000 pixels height - 16 MP image. Converted to PPM format, the space occupied by the image is 119 MB. Figure 6.10 contains the message that will be encoded.

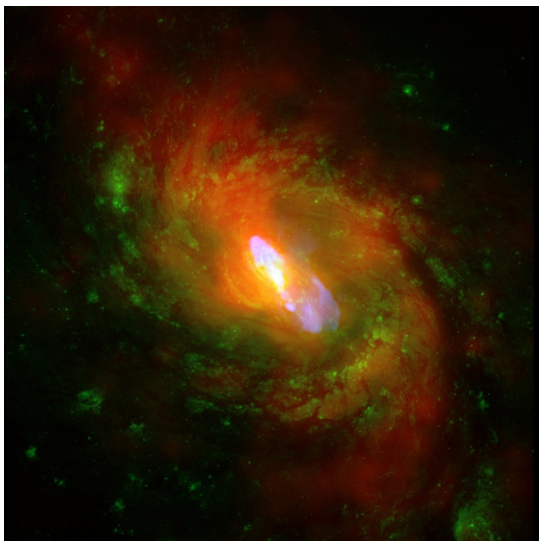

Figure 6.10: Stego message - image of a galaxy

Having such a large input file as stego message, I decided to use the roadmap

image described in the previous section as cover. Using 3 bits to hide data, it provided enough space to hide the complete input image. The final stego-cover can be observed in Figure 6.11.

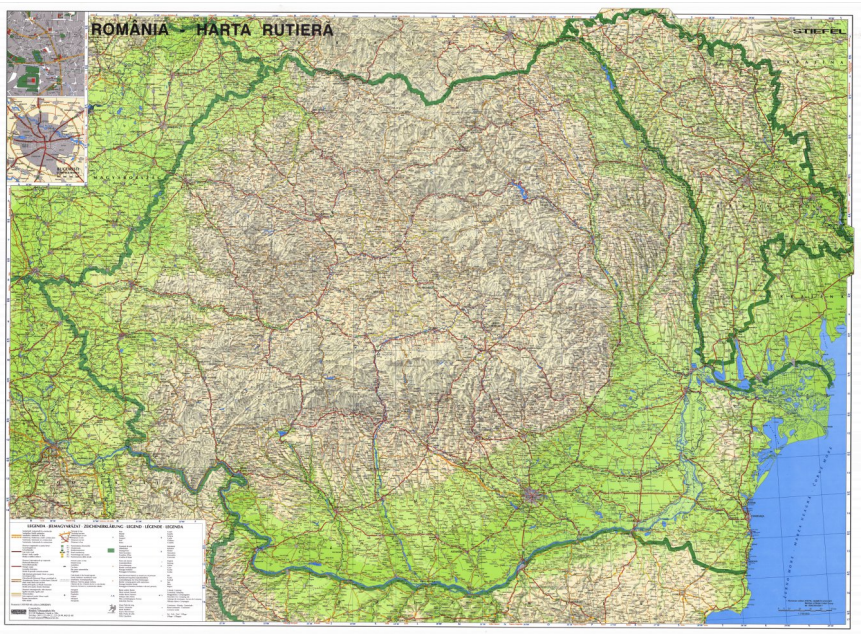

Figure 6.11: Final stego image

Chapter 7

Conclusions

The goal of this project was to implement an application That uses the LSB steganography method in order to hide data and also recover it. Because communication involves a sender and a receiver, there are two ways in which the application can run : as an encoder or as a decoder. For the encoding part the message is hidden into the least significant bits of a bmp image, thus resulting the stego-image. This image is then given to the decoder to extract the data that has been hidden.

There is a serial version of the algorithm, but also a parallel one. The parallelization was done using boss-worker model described in chapter 5. In order to implement the model I used PThreads. Both data hiding and data recovery can be done in parallel. As neither process depends on the number of threads, one can use 4 threads for encoding and 8 for decoding (or any other combination). The only variables that must be constant for both processes are the number of bits used from each color component and the size of the chunks in which the input image is divided in order to parallel encode/decode data.

In order to test the limits and performances of the application I used three different image sizes: 1.9 MP, 24.3 MP and 131.7 MP. The messages that were hidden into the input images were books downloaded from the Gutenberg Project website.

For the data hiding part the best performances were obtained with test cases number 2 and three. For the small image parallelization is not really justified, because the overhead introduced by the creation of boss and worker threads is too much compared to the time saved with parallel processing. This is why maximum speedup reached for this test case was 0.54 for 2 threads, continuing to decrease for a higher number of workers. For the last two test cases parallel hiding was more efficient - for the middle sized image maximum speedup was 3.86 for 12 threads and for the largest one the use of 8 threads led to a 3.85

speedup. However, in terms of efficiency the application did not excel for neither test cases - the highest efficiency value was 0.92 (third test case, 2 threads). In all situations efficiency was inverse proportional to the number of threads.

As expected, performances for data recovery were very similar to the ones corresponding to the first part of the application. The main reason for this behaviour is the fact that both parts of the application consists of the same three sections: read input image, hide/recover data, write output file/files. The second test case lead to the best results: 3.92 speedup for 12 threads. For the 1.9 MP image parallel data recovery was again slower than the serial version and for the largest image the use of 10 threads led to a 2.39 speedup.

Using very large images to only hide text files is not very suitable. For the last test case, even when using 1 bit from each color component the amount of available space to hide data is 50 MB. Text messages are usually shorter an therefore a smaller image could be used. Large images such the one containing the roadmap are perfect for hiding other images. The application can also hide images, as long as they are in ppm (ascii) format.

In conclusion, the main objectives of this project were accomplished. The result was an application that can be used in order to encode data into images and also to decode it. The amount of time the serial version takes was reduced to its quarter. The reason speedup did not reach higher values than 4 is the increased number of I/O operations. Data processing is not very time consuming, as it is reduced to bit operations, which are very fast. However, parallelization managed to reduce the serial time and if a suitable architecture for parallel processing is available, why not use it.

Chapter 8

Future work

One of the first improvements that could be brought to the application is adapting it for a parallel file system. This way each thread could write his chunk of the image directly to the output file, without further needing a bash script to concatenate pieces of files. I/O operations are currently the biggest "problem" for most applications that deal with large input or output data flows, but a parallel file system could solve the problem.

Another betterment would be modifying the application in order to work with different files, not only bmp images. Audio and video files are also great stego covers. In addition to this, the range of input files could diversify, containing other text and image image formats and even audio or video.

The current implementation of the LSB technique hides data in all three color components. However, this is not a must. The user could select the channel or channel that will be used for data hiding.

In order to better hide the data in case the message is too small compared to the input image, more complex algorithms could be used. Hidden bits can be dispersed throughout all the image using a unique key that only the sender and receiver posses.

While simple to implement, LSB hiding method is quite easy to detect. In order for the output stego images to pass steganalysis tests, the application could embed data only in certain regions of the image. Experimentaly it has been proven that data hiding into skin tone areas is less likely to be detected. The reason this happens is the fact that "embedding into these regions produces less distortion to the carrier image compared to embedding in a sequential order or in any other areas" [3].

One of the main advantages that steganography has over cryptography is the fact that the message is invisible to someone that intercepts data exchange between two persons. However, if an image is suspected of containing hidden

data, there are many steganalysis tools that could easily decode the content of the secret message. Most methods used for this purpose are based on statistical analysis ; one of the most used is noise detection. So, once a stego message is detected its whole content is revealed. This is where cryptography could come in hand. Using both steganography an cryptography one could create a powerful tool for data hiding. Figure 8.1 contains a schema of how such an application should work.

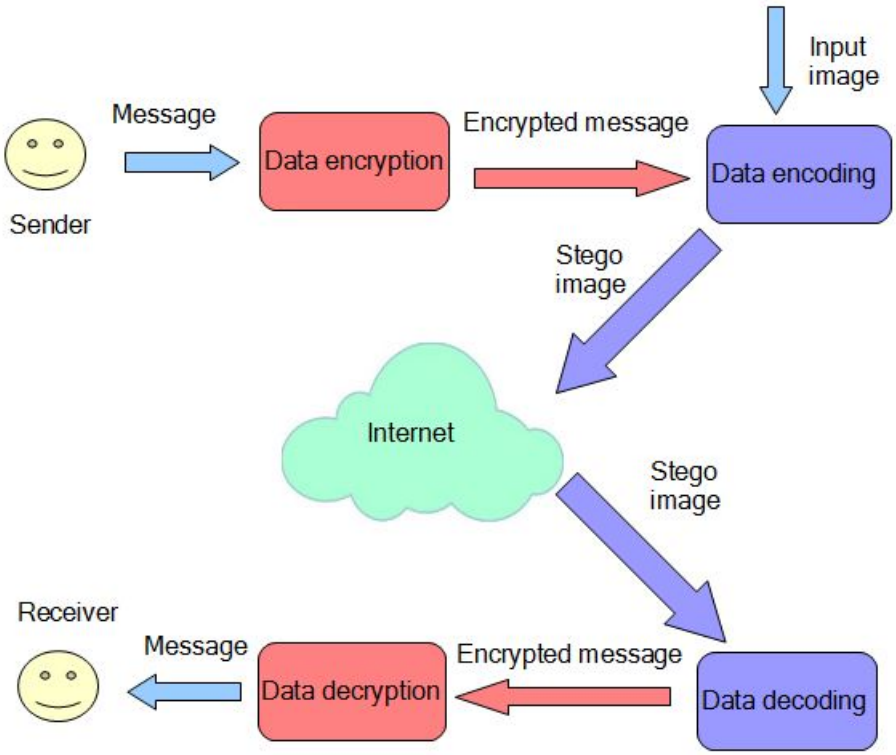

Figure 8.1: Communication using cryptography and steganography

Bibliography

[1] Tayana Morkel and Jan H P Eloff and Martin S Olivier. An Overview of Image Steganography. Information and Computer Security Architecture (ICSA) Research Group, Department of Computer Science, Univeristy of Pretoria, Sandton, South Africa. June/July 2005. [cited at p. 3, 5, 6, 7, 8, 9]

[2] Granthan, Beau. Bitmap Steganography: An Intoduction. April 1997. [cited at p. 3, 13, 14]

[3] Cheddad, Abbas. Steganoflage: A New Image Steganography Algorithm. University of Ulster, September 2009. [cited at p. 3, 7, 9, 40]

[4] Johnson , Neil F. and Jajodia, Sushil. Exploring Steganography: Seeing the Unseen. April 2007. [cited at p. 3, 14]

[5] Provos , Niels and Honeyman, Peter. Hide and Seek: An Introduction to Steganography. January 2004. [cited at p. 4]

[6] Krenn, J.R. Steganography and Steganalysis. January 2004. [cited at p. 3, 4, 6, 9, 10, 11]