UNIVERSITATEA BABEȘ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA: INFORMATICĂ ROMÂNĂ

LUCRARE DE LICENȚĂ

Steganografie în mediul digital

Absolvent:
Pricope Ștefan-Cristian
Profesor îndrumător:
Dr. Suciu Mihai, Conferențiar Universitar

2020

Abstract

Steganography is the science of concealing a piece of information within another piece of information without affecting the latter in a noticeable way and therefore alerting any intruders of the existence of the former. This thesis presents both existing and new ways of embedding computer files and data into different digital multimedia formats as covert as possible while still allowing the encoded information to be retrieved at a later time without any significant losses. It also presents the structure of some of the most common multimedia files that are used in the modern day digital world and are viable candidates for the role of the cover file in the steganographic process.

Contents

Chapter 1

Introduction to computer steganography

The practice of hiding and securing information and messages between different parties has played a major part alongside human history, especially during times of war when it was vital that the enemy didn't intercept the strategies and even if they did, they would have no idea what to do with them and would have to dedicate a lot of time, money and personnel to decode the communications. Two of the most famous manifestations of this practice are cryptography and steganography[14].

Trying to differentiate between cryptography and steganography is not difficult, the only thing they have in common is their end goal - making sure that a piece of information sent from one place to another is secured and that only the right recipient will be able to read and understand the received information. The difference lies in the methods they use and the time it takes to reach that goal.

Cryptography focuses on altering the information itself, encrypting it using a key only known by the receiver and sender[1], making it harder for any possible meddlers to alter the meaning of the message or even just understand it. Steganography is more concerned on hiding the fact that there even is any information being transmitted usually by embedding it in something else (hereby referred to as a cover), thus if any interlopers were to actually look at the cover, they wouldn't even be aware of the fact that it contained secrets.

In other words, both methods are meant to be used over an open and unsafe environment, and while cryptography tries to hide the contents of the message but not the fact that there is a message being sent, steganography tries to hide the communication altogether. The main advantage of these 2 methods is that they are not exclusive, they can be used together for maximum safety of the information.[2]

With the evolution of the Internet and the increasing usage of personal computers in day to day activities we needed to secure the data sent over the network between the users. The efforts were lead by cryptographers who developed thousands of algorithms for encrypting the information and the very few remarkable ones are still being used[3]. But that doesn't mean steganography was left behind, researchers developed plenty of new algorithms for hiding information using digital files as a cover and the Internet as the transmission environment.

In theory all types of digital files can be used as cover - shared libraries and executables can be edited to include the hidden data and then be recompiled/rebuilt, text files and documents could be modified to enclose the information in secret places/paragraphs, and multimedia such as images, music and video files can be changed to carry the digital data into their metadata, pixels, audio frequencies, motion vectors and much more. This paper will go into details about some of the most popular formats used for multimedia files and showcase their internal structure and document techniques used in steganography.

[1]This is only true in symmetric cryptography and is a gross oversimplification of cryptography as a science, but it is just meant to get a point across and help differentiate between the 2.

[2]The speed of encoding/decoding the information is greatly decreased when these 2 are combined which is the reason that nobody merges them, preferring to juse use encryption to keep the information safe.

[3]The reason cryptography is in the spotlight is because it is much faster, mathematically proven, and it hides all the data without leaving anything in plain sight(like steganography does with the cover).

Chapter 2

Steganography methods

2.1 Least Significant Bit(LSB) Insertion

2.1.1 Sequential

Least Significant Bit or LSB is by far the most used method when talking about any type of steganography. Given that the smallest unit a computer can understand and process is usually a byte, altering only the least significant bit will not change the transmitted information in a noticeable way to any external parties. It is much easier to showcase what a byte contains and what the LSB change implies and how it works. A byte contains 8 bits, so this means that the values a byte can take range anywhere from 0 to 255 (inclusive)[1]. Let's assume that we have an array of 4 random values in consecutive memory : 217, 127, 100, 62 (all values are in decimal), each stored on exactly one byte, and that we want to hide our grade in Numerical Analysis from our parents (in this case a 3) using a LSB substitution.

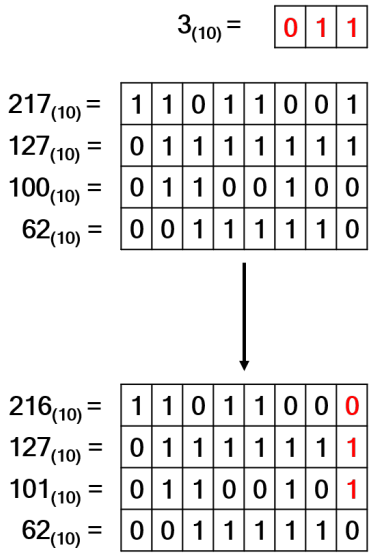


Figure 2.1: How the sequential Least Significant Bit change works

As we can see from Figure 2.1, we have succesfully altered the least significant bit of the first 3 bytes of the stream in order to hide our grade : 217 became 216 when we changed the last bit from 1 to 0, 127 was unchanged because it already had the last bit set, and the third byte became 101 after toggling the final

[1]This is the case for unsigned bytes, but given that we are talking about a method that only deals with the least significant bit, we can safely ignore the most significant bit, also known as the sign bit.

bit. Furthermore, the rest of the stream (the fourth byte, 62) was not affected because we already hid the entirety of our secret message. While this is great because we only hide exactly as much as we need and not a byte more, we have a high risk of corrupting the hidden message in case our cover image gets compressed or loses even a single byte when sent over a network. Basically, we are trading data redundancy in order to get simplicity and efficiency.

Sequential LSB insertion is the simplest and most common way of embedding any kind of information into a byte stream that is then shared. It has been thoroughly discussed by a great deal of researchers and has been the subject of many papers where it was analyzed and benchmarked[3][15] . Being the most popular technique also means that any flaws the method has are widely documented and showcased. Steganalysis[2] performed on outputs created using this algorithm has shown that it is unreliable to stay undetected if an outside party intercepted the message containing the cover file[22]. Furthermore, doing a simple reverse engineering on the algorithm reveals even more issues with this naive encoding : if a single bit that is part of the secret message was flipped from the cover file byte stream, the message would become corrupted and the original secret would be lost forever. This means that sequential LSB insertion is not resistant at all to any form of lossy compression where some of the original data may be lost in order to reduce the used disk space because it would lose most, if not all, of the embedded file information.

Furthermore, it is extremely easy to compute the carrier storage, i.e. how many bytes we are able to hide into the cover file or in other words, the maximum size of the secret message that we can succesfully embed without losing anything while still keeping a covert profile. Assuming CDS or Cover Data Size (how many bytes are actually used to store the pixel information, no metadata information or chunk headers or anything like that), then the MMS or Maximum Message Size would be equal to

$$MMS = [CDS \ / \ 8] \ bytes$$

or the integer part of CDS divided by 8. This should come as no surprise since sequential LSB is altering 1/8 (an eighth) of each byte when embedding a bit of the secret information so the maximum capacity makes sense to also be 1/8.

2.1.2 Scrambling

As documented earlier, sequential LSB insertion algorithm has a decent number of flaws so it was needed to develop some new techniques that are not relying as much on the cover file not losing a few bytes or undergoing a compression algorithm. In other words, it needed to introduce a few redundancies to ensure that the secret message wouldn't be lost as easily and that the message was not written in a sequential and direct order. They achieved this by not just changing the least significant bit in a sequential order, but by writing in an apparently random order (in simpler terms, scrambled) that could be reproduced by having the right key or by using the same algorithm in order to retrieve the embedded information. In this subsection we will discuss a few of the most common scrambling techniques and introduce a new one as well.

The most popular methods used for scrambling secret messages into various covers usually choose to simply ignore the entire data stream and only focus on a specific subsection and choose that as the carrier environment. After a smaller subsection is chosen (it can still be the entire actual data part of the cover, it's not an actual rule), we will have to generate the order in which we will write the message information. As mentioned before, this is derived using a key known only to the sender and the receiver that can be shared between the 2 parties using another transmission environment, preferably one that is encrypted and safe. The main logic is to use that passkey as a seed in a valid pseudo random engine when generating the order so that there are no collisions and that only the right key will produce the right order.

There is also an option for when there is no safe method of transmitting a key and that is to scramble the secret message within a certain order that is not sequential. But as long as the receiver and transmitter have no way of communicating the algorithm used in embedding the message (can also assume this because they have no way of sharing the key), this option is useless but is still interesting to look into because they are a variation of the other mentioned option. Having no key to generate the order makes this option easier to showcase(since we don't have to also simulate a pseudo-random engine and a seed). Let's assume we have

[2]Steganalysis is the study of steganographic methods, including but not limited to : differences in file sizes or in color histograms, secret message redundancy, embedding capacity and performance etc.

a byte stream of random data[3] and we want to again hide a grade from our parents, in this case a 6 (or 110 in binary). Instead of hiding the grade sequentially, the bits will be hidden into the last bit of every third byte and it will end up looking something like this :

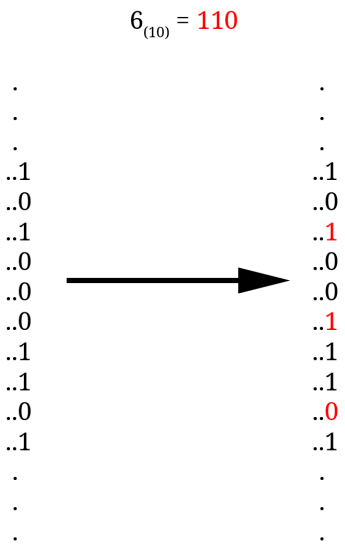


Figure 2.2: An example of scrambling LSB insertion

The key difference from sequential LSB insertion is that as long as the sender and receiver are aware of the used algorithm, any external parties will not be aware of the secret embedded message. Furthermore, this method has proven very useful because there are infinite ways of scrambling a message into the cover file without alerting any possible intruders and it ends up being an extremely hard guessing game for the attackers in their goal to extract the information.

The carrier capacity appears initially to be the same, but it is very important to remember that the scrambling algorithm used will usually work on a subset of the cover data bytes, not the entirety of it (just like in figure 2.2 we used only each third byte to store the information). Using the same notations as in the previous chapter we get that

$$MMS \leq [CDS \ / \ 8] \ bytes$$

so in most cases, the scrambled MMS will be smaller than the sequential MMS. However it is very important to note that this decrease in size comes with a great increase in data security and message safety.

This paper also introduces a new type of scrambling algorithm created by the authors that only works on lossless image formats that do not use any kind of interlacing when rendering the picture. It relies on scrambling the secret message into the image sub-blocks in a specific order that can only be deduced by having the right passkey. More information on this method is presented in chapter 3.2.1 after the introduction of image basics.

2.2 Metadata encoding

The word metadata was formed from combining the word "data" with the prefix "meta-" and is used to describe a special type of data that has information about other types of data[20]. In simpler terms, it means "data about data" and it keeps the same meaning in the digital world. It is much easier to visualize and understand the concept of metadata using a simple example : let's take a picture stored on a computer. We can see how it looks in figure 2.4.

[3]Now the actual data meaning can be safely ignored because we only work on the LSB and we have seen in the previous chapter it does not alter the data in a significant way such that an intruder might notice something is wrong.

```
sleshwave>exiftool goose.jpg
ExifTool Version Number         : 10.80
File Name                       : goose.jpg
Directory                       : .
File Size                       : 16 kB
File Modification Date/Time     : 2020:04:21 19:44:09+03:00
File Access Date/Time           : 2020:04:21 19:44:09+03:00
File Inode Change Date/Time     : 2020:04:21 19:44:09+03:00
File Permissions                : rwxrwxrwx
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Resolution Unit                 : inches
X Resolution                    : 72
Y Resolution                    : 72
Image Width                     : 1000
Image Height                    : 750
Encoding Process                : Progressive DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 1000x750
Megapixels                      : 0.750
```

Figure 2.3: Example of using a tool to read file metadata

Figure 2.4: A simple image file

As noted in the introductory chapter, every file can be seen as a byte stream. However it is very important to keep in mind that those bytes don't represent only the image data, the pixels seen on the screen, they are much more. They also contain information about the camera used to take the photo, the location where it was taken, when the file was created, when it was modified, etc. All of the aforementioned information forms the metadata. It varies from file format to file format where they store this information in the byte stream, how many bytes are allocated for each piece of information or if it even has any effects on the actual file data. Most of the time metadata fields are only parsed by the renderer software and are mostly hidden to the user, but there are a few ways of viewing the information:

- using a hex editor to view the raw bytes of the file and then mapping those bytes to the publicly available file format specifier - an international approved paper which specifies the meaning of the bytes in the file binary stream
- using the operating system to view more properties about the file, not just the actual data interpreted and displayed to the end user
- using a third party tool which already knows the mapping and meaning of each sequence of bytes in the file binary stream and can succesfully parse the metadata

The focus of this sub-chapter will be on the metadata fields that don't necessarily have any important effect on the data representation and theoretically could be altered, such as any comments from the author or any contact information. In the case of an image, changing important metadata fields such as the width or height of the picture are not very covert methods of embedding any information at all, proving that not all fields are equally important. We are left with the more *useless* metadata fields, but the good news is that most file formats allow these fields to have a variable size which is perfect for any steganographic purposes because it removes the size constraint of the secret message. While in theory this allows for ∞ MMS, there are a few limitations in place that significantly reduce that number:

- computers have a limited amount of storage space, in most modern day computers that would be about one terabyte. This means that the MMS will certainly be smaller than that.
- most file formats that allow metadata fields to be embedded in the byte stream of the file by an external party also have a sequence of bytes that indicates how long that metadata field is (how many bytes it contains). In most cases that sequence is 4 bytes long so this usually results in a MMS of $2^{4*8} = 2^{32} \approx 4.29$ *gigabytes*.

- in the pursuit of keeping the existence of a secret inside the cover file as covert as possible, the MMS is again limited by the CDS. This happens because the size of the cover file is almost always displayed to any parties without any interactions and it needs to not raise any suspicions or attract unwanted attention to the actual contents of the file. To give an example, it would be very weird to see a simple image in FULL HD resolution have a size of two gigabytes and will almost certainly alert any intruders. It is extremely hard to say an exact MMS based on this limitation because it depends on the original cover file size and it should be relative to that number. It is recommended that $MMS \leq 50\% * CDS$.

In the end it is important to note that metadata secret encoding is the most trivial method of embedding secret information into different kinds of cover files. However, this method comes with a great cost: almost every single tool that specializes in extracting metadata will be able to detect and identify our message without too much hassle because the final cover file still has to respect the format specification.

```
sleshwave>exiftool -s -Comment goose.jpg
Comment                         : Meet me tonight at 10PM
```

Figure 2.5: Example of message hidden in a file in the metadata section

2.3 Unused space embedding

Usually most file formats and even internationally approved and used protocols such as the TCP/IP have unused bits or bytes in their composition, either for future proofing the format in the form of reserved space or simply to serve as padding space mainly present to help the associated parsers.

One well known example of this is found in the BMP file format which appears only in the cases where the width of the image is not a multiple of 4. The standards specify that in this situation there needs to be added the right amount of bytes to fill so that the width will be divisible by 4. So if an image has a resolution of 1920x1080 (the width is the first number), there won't be a need for any padding bytes because

$$1920\%4 = 0$$

On the other hand, if the image had a resolution of 125x100, we would instead have

$$125\%4 \neq 0 \implies 4 - (125\%4) = 4 - 1 = 3\ bytes$$

to serve as a padding for each pixel line, ending up in a total of $100 * 3 = 300$ bytes of simple, raw, unused space that most BMP parsers will ignore because they know that data is meaningless.

Another very good example is in the case of the IPv4 packet header which has a reserved bit in the flags subsection, or to be more precise, the 49th bit. According to RFC3514[6] it is meant to be a security bit representing the true intent of the packet (GOOD or EVIL), however it is important to note that it was an April Fools informational memo and it is most likely that altering this bit will still go undetected by any kind of firewall or intrusion prevention system.

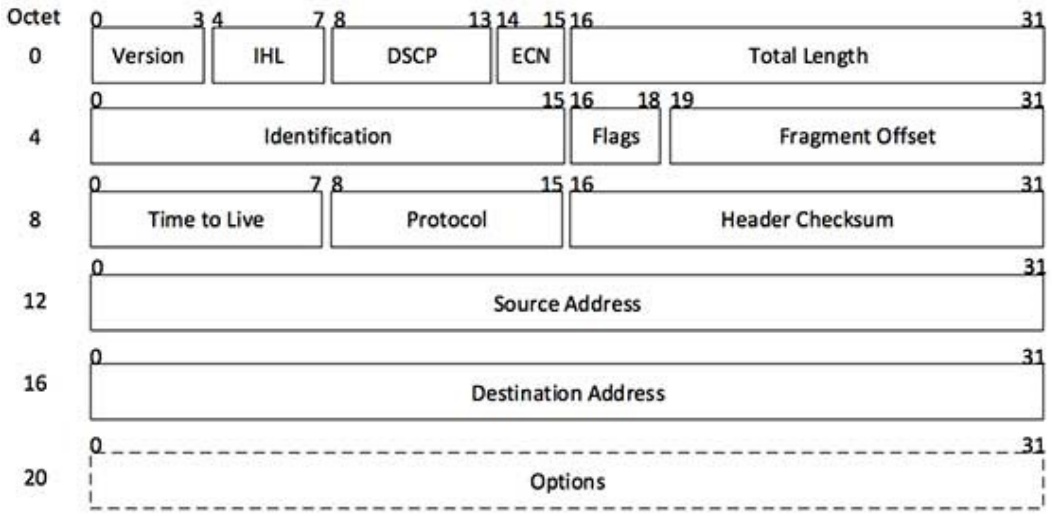


Figure 2.6: The structure of an IP packet

Given the IP packet header specifier represented in Figure 2.6 it becomes trivial to alter the first bit of the flags section to the current bit of a secret and then over multiple sent packets to actually send the message in one of the most covert ways possible. Furthermore, it is very important to note the contribution done by Kamran Ahsan and Deepa Kundur in their 2002 article entitled "Practical Data Hiding in TCP/IP"[12] because they also achieve the ability to send covert information by taking it one step further: using IP packet headers that may actually even be in use and by using Chaos Theory in the values of the Identification field sent over.

However, there is still one very important spot left in a file's byte stream where the secret message could easily be hidden: at the end of it. Most file formats either have specially crafted headers that announce the end of the file[4] or have a few bytes reserved at the beginning of the file that mark the length of the data[5]. It is important to note that both of these methods basically produce the same result technically speaking: a superior margin, a maximum amount of bytes that are to be read and interpreted by the parsers that contain the data relevant to the current file. After that region has ended, it is a free-for-all memory territory for secrets to be embedded. The same limitations from the previous chapter still stand, it is recommended that $MMS \leq 50\% * CDS$ in order to remain as covert as possible while still exfiltrating data.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
0009B8A0   ED C2 FF FB C5 F8 C5 A3 C8 B9 8D 98 32 A8 FA DC  íÂÿûÅøÅ£È¹.˜2¨úÜ
0009B8B0   D2 28 41 38 C7 52 D0 15 3A 3B F4 8B E1 A2 B3 4E  Ò(A8ÇRÐ.:;ô‹á¢³N
0009B8C0   A2 B3 A3 17 9B B0 60 93 51 8A 73 D9 73 47 9D ED  ¢³£.›°`"QŠsÙsG.í
0009B8D0   3F 5E F5 E2 47 F4 1C 0C F0 39 5C FF 24 37 3E EB  ?^õâGô..ð9\ÿ$7>ë
0009B8E0   17 FD 9B 37 C4 59 D9 E8 A7 78 15 91 40 8C F0 06  .ý›7ÄYÙè§x.'@Œð.
0009B8F0   A2 D3 FF 03 D4 FE E5 11 BA A8 95 1C 00 00 00 00  ¢Óÿ.Ôþå.°¨•.....
0009B900   49 45 4E 44 AE 42 60 82 68 65 6C 6C 6F 20 77 6F  IEND®B`‚hello wo
0009B910   72 6C 64                                        rld
```

Figure 2.7: Simple example of hiding a note after a file has ended

[4]One good example is the PNG format which has an IEND header and a couple of other information to mark that the image data has ended and that there is nothing left that is relevant to the parser. More information regarding this format is available in chapter 3.2.

[5]This method is found in BMP or WAV file formats usually, it is very simple and easy to work with for parsers. More details in later chapters.

Chapter 3

Image file formats and steganography techniques

3.1 Introduction

An image is a two-dimensional representation depicting any possible subject conceivable by human imagination, captured using an optical device (such as a camera or a telescope) or a natural object (human eyes). The image can then be rendered and displayed for other people to see either manually (by painting, carving etc.) or automatically (by using a computer). In this chapter we will focus on images captured using digital optical devices that are rendered automatically. The correct term for them is digital images, but throughout the rest of the paper they will be reffered to as images for convenience.

Figure 3.1: Lenna - Classic example of a digital image

Computers are programmed to do operations in a clear sequential way and this rule doesn't change when working with pictures. In order for a computer to be able to render an image, it needs to know some general metadata information about the photo, such as the width and height, as well as the data bytes of the image. These bytes are the actual representation of the picture which compose the two-dimensional pixel map[1]. A pixel is the smallest unit that a computer monitor can read and display. The pixel color is the result of merging the different color channels which compose the picture (such as RGB, YUV, YCbCr etc.). Here is an example of the entire process - let's assume that from the image data bytes the first 3 bytes have the decimal values 20, 127, 250 and that it uses the RGB color model. This means that when the computer will have to render the image, the first pixel will have the red component equal to 20 (0x14), the green equal to 127 (0x7F), and the blue equal to 250 (0xFA), in what will finally be interpreted as #147FFA by the monitor (variation of light blue).

[1]This is true for a lossless format, where each pixel is stored in memory. It is not exactly the case for lossy formats such as JPEG where the image goes through processing before being rendered. More information later in the chapter.

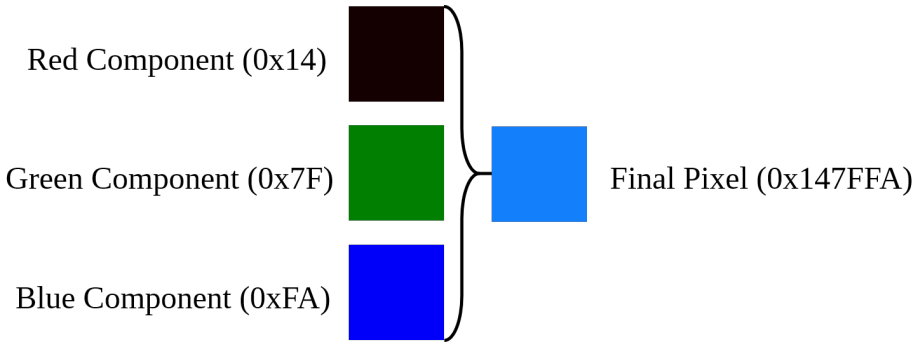


Figure 3.2: How 3 colors channels build the pixel

By merging multiple pixels over a two-dimensional space, they will eventually start to resemble an image that can be stored on a disk as a byte stream and can be rendered anytime by parsing the aforementioned stream. Each pixel can be represented as a point in that space with coordinates that are part of the unsigned integer domain. An entire row of pixels, i.e. pixels that have the same ordinate value, is sometimes also referred to as a scanline because in the earlier days of modern computing, computers would be given the width of the image and based on that value they would read a precise amount of bytes and render it on the screen before moving on to the next scanline, repeating this process until there would be no more information.

It is important to note that most of these developments have been done in a time where the maximum storage was extremely limited and not very fast, very different from what it is today. In order to save some space they looked into different compression algorithms to apply to the byte streams and today they fall into two distinict categories: the lossless algorithms are the ones that compress all the original information without destroying any of it, while on the other hand there are the lossy algorithms which are able to identify which information is useless and delete it accordingly. This concept also applies to file formats and we will see in later chapters more concrete examples.

With all of this information in mind, we can now procede to discussing the most commonly formats commonly used in modern day multimedia.

3.2 BitMap Picture (BMP)

The BMP file format, also known as the device independent bitmap file format(DIB), bitmap image file or just bitmap, is a lossless[2] image file format originally designed by Microsoft back in 1986 in order to store two-dimensional digital images on their Windows operating system. Over the years it has developed plenty of variations and extensions that were based on the original specification but this paper will focus only on the most common available ones, no extended versions that are looking to improve the format since they do not add anything interesting or new to the way the format stores the data thus affecting the steganography algorithms.

As with almost every file format, the final BMP byte stream can be seen as a result of the merge between the BMP header which contains metadata about the file and the BMP data which is the actual pixel information. As we can see from table 3.2, the BMP header stores a lot of important information about the image that is useful for any rendering software while making sure to allocate enough memory to be able to display the picture on the screen and other essential steps. It is also important to note that all the structures seen in the BMP header use the little-endian format for representation and are usually more troublesome on the systems that have the default set as big-endian.

[2]It is true that the format specification standards supports compression but further research reveals that currently it only supports lossless types of compression, such as the Huffman or Run Length Encoding algorithms.

Information	Size	Offset	Description
Signature	2 bytes	0x00	Two chars, 'B' and 'M'
File size	4 bytes	0x02	Total file size in bytes
Reserved	4 bytes	0x06	Unused space
Data offset	4 bytes	0x0A	Offset to get to the actual BMP data
Size	4 bytes	0x0E	Size of the left header information
Width	4 bytes	0x12	Horizontal size of the image
Height	4 bytes	0x16	Vertical size of the image
Planes	2 bytes	0x1A	Amount of image planes
Bits Per Pixel	2 bytes	0x1C	How many bits are used to represent each pixel
Compression	4 bytes	0x1E	Indicates the type of compression used
Image size	4 bytes	0x22	The size of the compressed image, can be 0
X pixels per Meter	4 bytes	0x26	Horizontal resolution in pixels/meter
Y pixels Per Meter	4 bytes	0x2A	Vertical resolution in pixels/meter
Colors Used	4 bytes	0x2E	Number of actually used colors (based on Bits Per Pixel)
Important Colors	4 bytes	0x32	Number of important colors (usually all)

Analyzing the obligatory BMP header fields we realise that most of them are useless for any steganographic purposes mainly because they can't be altered without having major consequences on the renderer software but there are still a few interesting ones left:

- The 4 bytes that are reserved and unused could be very well put to use by using the methods presented in chapter 2.3, so we can use either this space to send parts of a message over multiple BMP files, or we can store the secret message size in these 4 bytes and write the secret after the actual image data has ended.
- Data offset could be useful because some renderers use this field to indicate the offset of the actual image data and just skip any other irelevant metadata information, allowing us to hide information in those fields.
- The width and the height of the image usually are altered to hide bottom parts of the image that may contain hidden information or by masking the result of the merge of two images and just showing the top one. Let's take for example this image of the sun that is obviously missing a part for demonstration purposes.

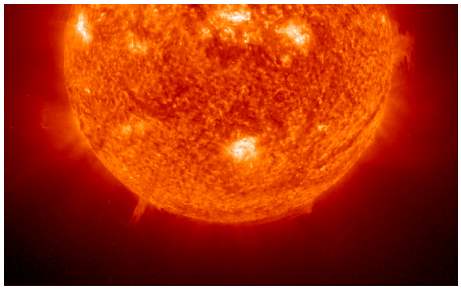


Figure 3.3: Incomplete sun image

However if we adjust by trial and error the height of the image to see if there is any more bytes to render, we would notice that there really is a message hidden with those bytes that some steganalysis softwares would detect but any renderer software such as the Windows Media Viewer would always miss. In other words, we have tricked the software to not display more scanlines than we wanted it to display, even though the information is there, not corrupted in any way.

Figure 3.4: Complete sun including the hidden message

Moving on from the metadata block of the BMP format to the actual data byte stream, we find additional interesting information about how the pixel data is actually stored. Most images use the Red, Green and Blue also known as RGB values to compose the final pixel color, but for an unknown reason the creators of the bitmap format decided to store the information in the reverse order in the data stream as Blue, Green, Red or BGR. However this is not the only change they made from the other common image formats available at the time: rather than storing the scanlines from a top to down order, BMP decided to store them bottom-up.

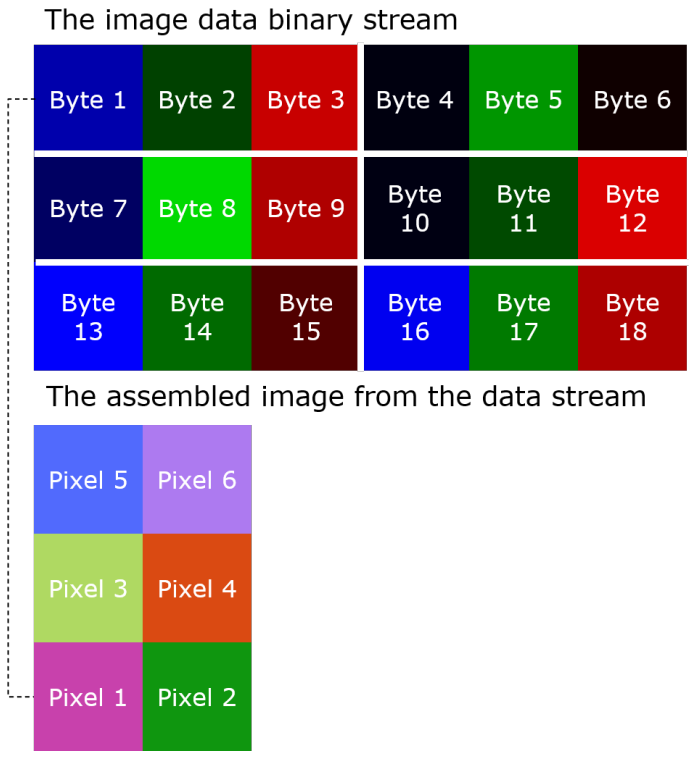


Figure 3.5: How the BMP image is rendered

This behaviour can be noticed in figures 3.3 and 3.4 where the top part is cropped in order to hide a message and we can now understand why it is only that part that can be made redundant in the steganography

process: because the last bytes of the image data binary stream are actually responsible for the rendering of the top section of the picture. Seeing and understanding how the carrier format works is the first and most important step in embedding messages in a covert way.

3.2.1 Image sub-block scrambling using the BMP format

Sub-block scrambling is a type of scrambling algorithm, like those introduced in chapter 2.1.2, that can be applied to images stored in the BMP format. It was created by the author of this thesis based on the idea of the JPEG format and the algorithm it uses during compression that is applied only on sub-blocks of 8x8 size. We mentioned in chapter 2.1.2 that most scrambling algorithms have

$$MMS \leq [CDS \ / \ 8] \ bytes$$

but the target was to design and implement an algorithm which can make use of all the bytes available in the cover, and we ended up with this method which has

$$MMS = [CDS \ / \ 8] \ bytes \tag{3.1}$$

making it similar in storage capacity to a sequential algorithm. That performance was achieved following a simple procedure:

- Separate the image into multiple blocks of BLOCK_SIZE (this is a global constant value which is usually equal to 8 but can be higher or lower). It doesn't matter if the width or height of the image are not a multiple of BLOCK_SIZE, the algorithm also works with non-square blocks.
- The algorithm needs a password to work with in order to use it as a seed when generating a permutation. Each block calls a helper method to obtain the given permutation for itself based on its own width and height. That permutation is the order in which the data will be written.
- The first block is reserved for writing the metadata about the message(such as the length of the secret) and the other blocks are for writing the actual contents of the message.

```
global constant BLOCK_SIZE;

procedure embed_subblock(secret, subblock, random_engine)
  #Generate a permutation from 0 to subblock.width * subblock.height
  #using the random_engine given. That permutation will be the order
  #used when writing the secret data into the subblock data.
  writing_order←random_permutation(0,subblock.width*subblock.height,random_engine);

  for each byte_index in writing_order do
    #Getting the absolute index of the subblock rows and
    #columns where we need to write the secret information.
    subblock_row←subblock.starting_line_index+byte_index/subblock.width;
    subblock_column←subblock.starting_column_index+byte_index%subblock.width;
    set_least_significant_bit(subblock.data[subblock_row][subblock_column],secret);

    #We move to the next bit of the secret information
    advance_to_next_secret_bit(secret);

procedure pseudo_scramble(cover, secret, password)
  #Creating a new pseudo-random engine with a given seed.
  random_engine←new pseudo_random_engine(password);

  for each subblock of cover with subblock.size≤BLOCK_SIZE
    if subblock = cover.first_subblock
```

```
    #Used for hiding the size of the secret into
    #the first subblock of the cover in an order
    #given by the pseudo-random engine random_engine.
    embed_subblock(secret.size, subblock, random_engine);
  else
    #If it's not the first subblock of the cover,
    #write the actual secret into the subblock.
    embed_subblock(secret.iterator, subblock, random_engine);
```

Listing 3.1: Pseudocode of the scramble algorithm

It is much easier to explain the algorithm using a simple example, so let's begin with the image represented in figure 3.6. It is BMP image with a resolution of 10x6 pixels (10 pixels wide, 6 pixels tall). Furthermore, let's also create the parameters needed by the algorithm in order to embed a message:

- The secret message we want to hide in the cover file is "Hello World", no quotes.
- Given the aforementioned message, we can calculate its size: 5 characters for the word "Hello", 5 characters for "world" and 1 character for the space inbetween. Total message length is 11 characters or 11 bytes using the ASCII standard.
- The password used for generating the seed is "super_secret_key", no quotes.
- The BLOCK_SIZE used in splitting the original image is equal to 4.

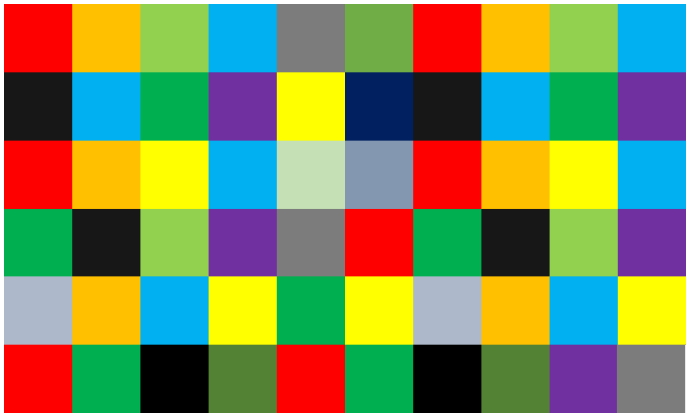


Figure 3.6: Original BMP cover image

Knowing all of this information we can already know if the cover file is going to be able to hold the secret message based on the formula 3.1. We know that the BMP format uses 3 bytes for storing each pixel and that our image is 10 pixels wide and 6 pixels tall, so we have that

$CDS = 3 * 10 * 6 = 180\ bytes \overset{(3.1)}{\Longrightarrow}$

$\overset{(3.1)}{\Longrightarrow} MMS = [180/8] = 22\ bytes$

Since our total message length is smaller than MMS, we know that we will not be running out of cover space during the embedding process. After making these verifications, we can begin by initializing our pseudo-random engine based on the given password meant to be used as a seed and we can also start splitting the original image into sub-blocks, ending up with a result similar to figure 3.7. Since neither the width or height of the image are multiples of the BLOCK_SIZE constant, we have several smaller blocks (to be more precise, blocks 3, 4, 5 and 6) that are still going to be used for embedding purposes.

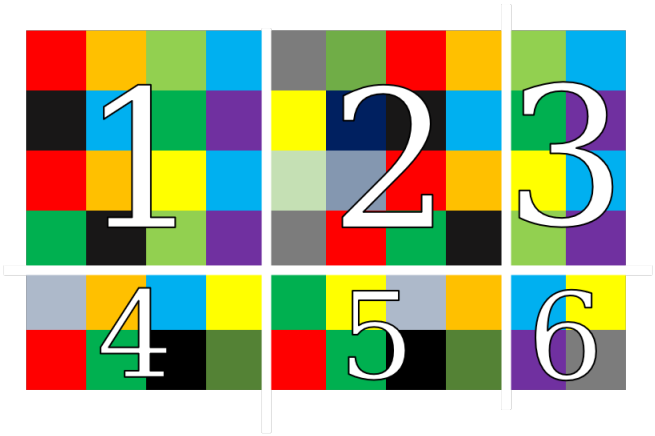


Figure 3.7: Sub-blocks division of the original image

We can now begin the actual embedding process. From the pseudocode presented in listing 3.1 we see that the first step is to embed metadata about the secret message, more specifically its length represented as an unsigned integer on 32 bits. But before we can do that, we must ask the pseudo-random engine to generate a permutation between 0 and $subblock.width * subblock.height$, excluding the last value. For this example specifically, we know that

$$subblock.width * subblock.height = 4 * 4 = 16$$

and let's assume that the engine returns us the following permutation:

$$(5, 10, 12, 3, 0, 15, 4, 9, 14, 13, 8, 2, 7, 1, 6, 11)$$

With this order known and with the fact that each BMP pixel is represented on 3 bytes giving us 3 least significant bits for embedding purposes in mind, the process of embedding the message length (11 or 0x0000000B) into the first subblock goes as follows:

1. 5th pixel is the first one, all LSBs are set to 0.
2. 10th pixel, all 3 LSBs are set to 0.
3. 12th pixel, all 3 LSBs are set to 0.
4. 3rd pixel, all 3 LSBs are set to 0.
5. 0th pixel, all 3 LSBs are set to 0.
6. 15th pixel, all 3 LSBs are set to 0.
7. 4th pixel, all 3 LSBs are set to 0.
8. 9th pixel, all 3 LSBs are set to 0.
9. 14th pixel, all 3 LSBs are set to 0.
10. 13th pixel has the Blue channel LSB set to 0, Green channel LSB set to 1, Red channel LSB set to 0.
11. 8th pixel has the Blue channel LSB set to 1, Green channel LSB set to 1, Red channel is unchanged.
12. The rest of the pixels are unchanged.

A visual representation of the order based on the given permutation can be found in figure 3.8. We can also notice that we managed to embed 4 bytes of information into a simple block of 4x4 pixels, or 48 bytes of total data, and we still have 2 bytes of free space based on 3.1(one bit available in 8th pixel and 3 bits in pixels 2,7,1,6,11).

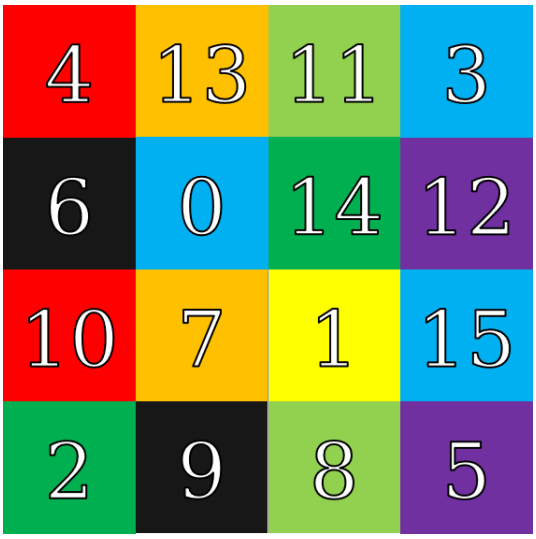


Figure 3.8: Order of writing into the first block

After embedding the metadata, we can begin hiding the actual message data into the subblock bytes. The sub-block size is 4*4 pixels total, which means 48 bytes in total space, resulting in exactly 6 bytes we will be able to write into this subblock (and every other block which uses 3 bytes to store a pixel and has 16 pixels in total). We start by converting our secret message in binary and we end up with something like this:

1. H = 01001000
2. e = 01100101
3. l = 01101100
4. l = 01101100
5. o = 01101111
6. (space) = 00100000
7. w = 01110111
8. o = 01101111
9. r = 01110010
10. l = 01101100
11. d = 01100100

By knowing we have 6 bytes available in the first sub-block we will be able to hide the string "Hello ", which is impressive because it is already more than half of our total message in an image that is only 4 pixels wide and 4 pixels tall. Before starting the actual embedding process, we must again use the pseudo-random engine to generate a new permutation ranging from 0 to the total amount of pixels in the block. Again for the sake of our example, let's assume that we get this permutation:

$$(10, 4, 12, 6, 1, 15, 9, 3, 0, 8, 13, 5, 2, 14, 7, 11)$$

Using the binary conversion of our message, we begin by taking 3 bits each time from the binary stream and embedding them into each channel of a pixel in the order given by the permutation. We can see what bits we have written into which pixels in figure 3.9 (the first bit is the one written into the Blue channel of the pixel, the middle one is the Green channel and the last bit is encoded into the Red channel).

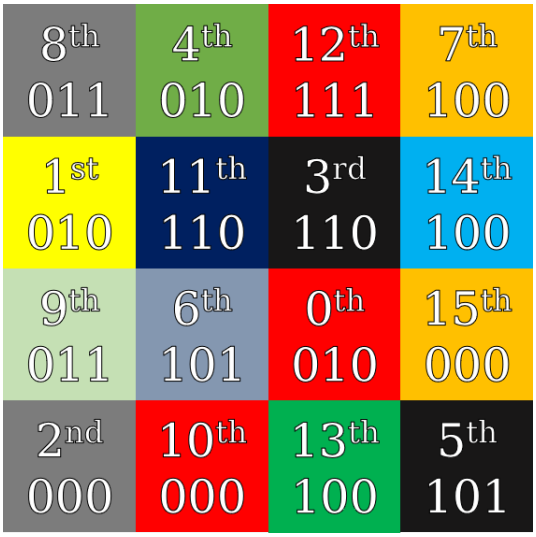


Figure 3.9: Order of writing the actual message into the second block

In the case of the third block the situation is slightly different because the width of the block is not equal to BLOCK_SIZE but this change does not change the behaviour of the algorithm at all. In this case we have a block that is 2 pixels wide and 4 pixels tall, ending up with a total of total 24 bytes needed for storage that can accomodate 3 bytes of secret information, or in our case the letters 'w', 'o' and 'r'. In this case we generate a permutation from 0 to 8, excluding the last value:

$$(3, 6, 0, 1, 4, 7, 2, 5)$$

By going through the same process as with the previous block we get the result that can be seen in figure 3.10. This leaves us with the target of hiding only two more characters: 'l' and 'd' in block number 4 which is identical to block 3 in size but was rotated by 90 degrees.

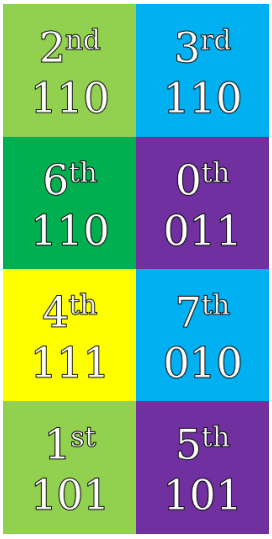


Figure 3.10: Order of writing into the third block

For the last time we generate a permutation using the engine in the same interval as above and we obtain:

$$(5, 3, 0, 7, 4, 2, 6, 1)$$

And by going again through the steps described above we manage to hide the last two remaining characters and we still have a byte free in the sub-block. The remaining blocks of the image remain unchanged and theoretically could be used for hiding other messages, but there are a few issues that can arise with this, the main one being message separation and identification, how each message is divided etc.

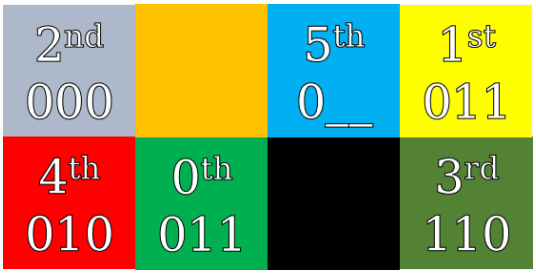
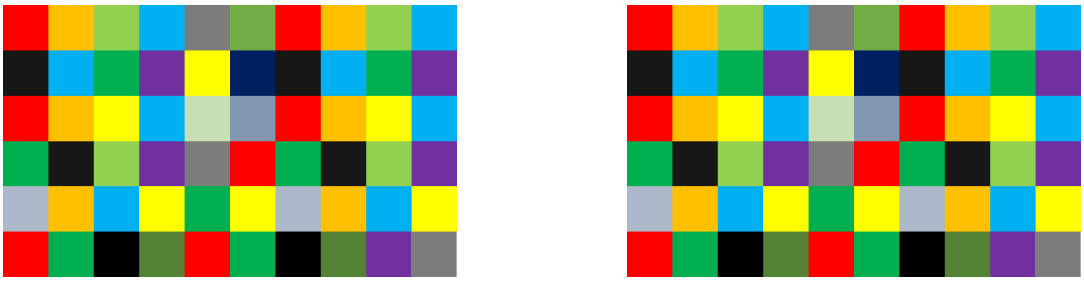


Figure 3.11: Writing the actual message into the fourth block

After the algorithm finishes and the modified cover image is saved, there is absolutely no difference to the human eye between the two pictures. This is the power of Least Significant Bit insertion when using digital images as the carrier format. The only difference between the two files is at a bit level, making it one of the safest and most covert ways of sending secrets over an untrusted environment.



(a) The original image

(b) The altered image containing "Hello world"

Figure 3.12: Comparison between the original and modified cover images

Like any other steganography algorithm, this method also has advantages and disadvantages. The main benefits of this algorithm are:

- **Security**. Choosing a BLOCK_SIZE of only 8, each normal sized block will have a total of 64 pixels and there would be $64! = 1.2688693e + 89$ ways available (just for a single block) for the algorithm to write the secret information. Given the fact that each block has its own permutation generated specifically for it based on a pseudo-random engine makes it very hard for any intruders to try and retrieve the secret message without having access to the key. Increasing the size of the BLOCK_SIZE only strengthens the overall security of the method while still keeping the same message capacity.
- **Efficiency**. The algorithm is extremely fast because it performs very few demanding operations and it is taking advantage of bit-wise operations. Furthermore, there is also the option of adding parallelization because there are no conflicts whatsoever between the parties involved, thus ending in an even more efficiency increase for the entire operation. There could be a small issue regarding the generation of the permutations based on a key in a multi-threaded environment because it may be prone to some implementation language issues with the pseudo-random engine.

However there are quite a few issues with this algorithm, issues that can render the method useless and make it obsolete in a faster than normal timeframe:

- **No data redundancy**. There is absolutely no mechanism in place in order to prevent the loss of data. Most communication environments nowadays usually compress the files before sending them over the wire and that compression may be lossless, but in most cases it is lossy. And even by sending the file in a way that it will not be compressed, there is also the slight change of altering and a small change in a single bit can and it will make the whole secret message corrupted without any method of recovery.
- **Detection via pattern identification**. Working only within small sub-blocks of BLOCK_SIZE means that we also limit the attack surface of any outside parties and allow them to focus on finding any details that can help them identify the key that was used when encoding the secret. For example, it is possible that an attacker will want to focus on the second block of the cover file and try to search for common extensions in the secret embedded file headers. If the intruder finds there is an order to the bytes of the block which produce that header that means that he is much closer to identifying the seed of the pseudo-random engine because he knows the second permutation generated by that seed and based on that information he may be able to deduct further permutations.
- **Limited market**. This algorithm was designed with file formats that are lossless, i.e. bitmaps that store the entire information of the image. While theoretically it could be used for PNGs as well, we will see in later chapters that PNG uses a mechanism called interlacing which makes it harder to work or even identify the sub-blocks of the image. Most modern online platforms have also moved away from BMP since they are far too big in size and the same image quality could be achieved by using other formats that have a much much smaller size, ending up in almost making the entire BMP format obsolete.

In the end, it is important to note that similar work has been done by a few researchers, the closest one being the work done by Hussein Al-Bahadili[4] in his paper where he chose to focus on scrambling the contents of the secret message using a similar password and seed generation technique before writing it to the carrier file. But while the 2 methods are similar in their goal, the way they reach that goal are different.

3.3 Portable Network Graphics (PNG)

Portable Network Graphics or PNG for short is a lossless image file format developed in 1996 that was designed to improve and replace the Graphics Interchange Format or GIF[18]. Similarly to the BMP format, it stores the entire information about the image in the data stream but the big difference between the two is that the image data in the PNG is always in a compressed format. Furthermore, another big improvement is the support for the alpha channel otherwise known as an "opacity" layer, allowing for transparency in the layers that build the image.

Figure 3.13: Example of a PNG file

As mentioned before, the alpha channel is one of the most important components of a PNG file. We can see the alpha channel in action in figure 3.13 where there are absolutely no other pixels in the image besides

those that compose the penguin (it is not a white background matching the color of the paper, it will be just the penguin regardless of the background color).

The structure of the PNG file format is simple and straightforward:

- The magic bytes at the beginning that are unique to the format. Every PNG file will always have these 8 bytes (also called magic bytes) at the beginning or it will be considered straight up corrupted by any rendering software:
  0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A
  Each byte in this sequence has a meaning but since altering even a single bit will render the file useless it means we will not be able to embed anything within these bytes.

- The rest of the PNG file is composed from a series of so-called chunks which contain both metadata and actual image data. Each chunk has a specific structure presented in the table below.

Chunk length	4 bytes
Chunk type	4 bytes
Chunk data	Chunk_Length bytes
CRC	4 bytes

  By changing the chunk length, chunk type or the Cyclic Redundancy Check(CRC) we will only manage to render the PNG file useless and get it flagged as corrupted by any decent renderer so the only logical remaining step is to modify the data found in the chunk data field if we want to embed secret information. However before we discuss that, it is important to note the most used chunk types present in a PNG file, their role and how the chunk data is structured inside them.

The minimum number of chunks in every valid PNG file is 3. There must always be at least an IHDR chunk , an IDAT chunk and an IEND chunk[3]. However the standard allows for a lot more types of chunks but they are usually non-vital during the rendering process, therefore getting the name of ancillary chunks. Going through the critical types of chunks we have:

- **IEND**, the simplest type of chunk. It is used to mark the end of the image and will always be the last chunk, signaling the renderer software that there is no more relevant information after this chunk. For this chunk the length will always be equal to 0, the type will be equal to the string "IEND" or 0x49454E44, the data field is empty and the CRC is actually calculated but will always be equal to 0xAE426082. Knowing the semnification of this chunk and how all software programs interpret it as the conclusion of the file, we see how easily it can tie into chapter 2.3, the one regarding writing secret messages after the actual file has ended. It is trivial to append bytes to the original file, but it can be harder to know how and where to separate the original data from the message, however having a clear boundary such as the IEND chunk helps considerably.

- **IHDR**, the header chunk containing critical image metadata. The standard specifies that this chunk must always be the first one in the file after the 8 magic bytes. The chunk always has the length equal to 13 or 0x0000000D, the type equal to the character array "IHDR" or 0x49484452. In the data field of the chunk we have the following structure:

Width	4 bytes
Height	4 bytes
Bit depth	1 byte
Color type	1 byte
Compression method	1 byte
Filter method	1 byte
Interlace method	1 byte

[3]There is also the PLTE chunk that must be present in some types of PNGs but it is so rarely found in any normal environment it is not worth being looked into.

And since the information in the data field will vary from file to file, the CRC is not a fixed value. While in theory it is possible to modify some of the aforementioned fields in order to hide something it is not recommended because altering even a single bit can alter the image in a significant way, thus losing the covert factor. Altering the width or the height of the image is even worse because it will cause a total corruption of the image since the software relies on those values to know when to expect the IEND field and if it is too early or too late in the data stream the parsing will most probably fail.

- **IDAT**, the chunk containing the actual image data. The image data is in a compressed format by default using the DEFLATE compression algorithm[7] on the stream obtained after applying a prediction method on the raw data of the image. To obtain the raw data needed for the steganographic algorithm we cannot just start writing the LSB of the compressed stream, we will have to decompress it and then reverse the filtering process that was used for each scan line. The first step to obtaining the raw image is to get the chunk data field in its entirety and to reverse the DEFLATE process in order to get the filtered image data stream. There are multiple types of this stream but the standard only defines the default one and marking it as 0 in the Filter method part of the IHDR data chunk. Each scan line of the image also begins with a byte representing the filter sub type indicating how each byte in that scan line is build relative to the surrounding bytes (important to note, the surrounding bytes not the pixels). As of the moment writing this thesis, there are 5 sub types recognized by the standard:

Filter byte	**Name**	**Value based On**
0	None	Raw byte values
1	Sub	The byte to the left
2	Up	The byte above
3	Average	Average of the left and the above bytes
4	Paeth	One of the 3: the byte to the left, the above or the top left one

A PNG file uses most of these filters because they are tied up to a scan line, not to the entire file. It is important to be able to reverse any of these filter types if they appear in order to get the raw R, G, and B values out of the stream to modify those. Modifying the stream before reversing the filtering process will only cause corruption to the image and any decoders will either warn that the file is malformed or will render it and any intruders will notice that it has been altered. However after the applying the defiltering method to the scan line and obtaining the raw data it is trivial to apply steganographic algorithms that rely on Least Significant Bit insertion. Since PNG is a lossless format, any modification done to the stream will be saved even after filtering and compressing the stream to save the new image containing embedded information.

Besides the critical chunks, there are also plenty of ancillary chunks that can be placed inside a PNG file without any software decoders complaining or malfunctioning. Users may also define their own types of chunks as long as they follow a simple protocol (having some special bits set or unset inside the 4 bytes associated to the chunk type). But there are already a few interesting chunks defined in the standard that are more than enough for steganography purposes:

- **bLob** can contain absolutely any sequence of bytes and it is guaranteed that no parsing software will try to render it so it is absolutely one of the best types of chunks that can be used to embed anything.
- **eXIf** is an interesting chunk because while defined by the most recent standard as a possible chunk, it does not have a structure defined, so it can contain any sequence of bytes, similar to the bLob chunk. If any software tries to parse it however, it would be interesting for eXIf chunks to contain data that would normally be found be associated with EXIF data such as the location or the time where/when the image was taken but to alter them to hide a meeting location and date.

- **tEXt** is also a good location for encoding messages, however it is the least covert out of all the aforementioned chunks because some decoders will use it to check for copyright, warning or disclaimers from the author before rendering the image. However since it does not have a maximum length, it can range anywhere from 1 byte to $2^{32} - 1$ bytes, making it a viable option for embedding hidden information.

In conclusion, given the fact that PNG is one of the most used carrier formats for digital images today, the ability to understand and hide messages inside this format is considered highly valued.

Chapter 4

Audio file formats and steganography techniques

4.1 Introduction

Sound is the result of a vibration created by a phenomenon that propagates through a transmission environment, ending up getting interpreted by our brain. However since this process happens in the physical world it is entirely analog so it would be impossible to store it on modern day devices which can only understand digital formats. Luckily the fast evolution of computers also brought conversion techniques in order to switch between analog sounds and digital sounds seamlessly, without any noticeable loss to the human ear. Using these methods we have gained the ability to store audio files in a digital format so it was only logical that several different file formats will be created to fit our needs. In this chapter we will discuss in greater detail about how the analog data is actually stored in the digital format and what are the most common extensions used for storing digital audio files.

We mentioned earlier that it is impossible to store analog signals in a digital environment. The solution to this problem is to convert the audio signal which can be represented as a continuous-time function into a discrete-time function using a process called sampling.

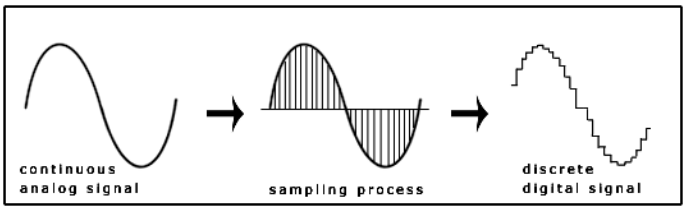


Figure 4.1: Converting a continuous signal into a discrete one[5].

The sampling process can be observed in figure 4.1. We can now introduce some new terms that we will use throughout the rest of the chapter:

- **Sampling rate** is the number of samples taken per second, or in other terms, how many discrete values we store for each second of the audio signal. The measurement unit for sampling rate is Hertz (Hz for short) and some of the most common values are 44100, 48000 and some of their multiples.
- **Bit depth** is the number of bits used to store a single sample after having it converted to a discrete value. The most common bit depths are 8, 16, and 24 which allow for 256, 65.536, and 16.777.216 different values.
- **Audio channel** is the term used to describe the sequence of bytes that represent sampled audio signals. An audio file can have multiple channels to better simulate the sound accuracy and origin in a

limited environment. Files with one audio channel are called mono, with two they become stereo and any more channels makes them surround. However, no matter the number of channels, usually all of them are equal in length and the samples from each channel are played simultaneously.

In the steganography field, the most common configuration that accepts alterations to the sampled data without losing any noticeable quality is a sampling rate of 44.1kHz with a bit depth of 16 and any number of audio channels, so this the ideal format that we will use throughout the rest of the chapter. The reason why this configuration is favored so much is because of the popularity that came with the invention of CDs and MP3s which used it as a default. Furthermore, any changes made to the sampled data are usually small enough that they will not be noticeable according to the Nyquist-Shannon sampling theorem [16], which is used to compute the condition such that the conversion from a continuous signal to a discrete one will capture all the relevant information. Using the aforementioned theorem and armed with the knowledge that the human physiology enables us to hear audio signals ranging from 20Hz to 20kHz, we can see why the 44.1kHz sampling rate is ideal in audio steganography.

4.2 Additional techniques used in audio steganography

4.2.1 Frequency domain steganography

So far in this thesis we have talked about what can only be classified as spatial domain steganography, like how a pixel of an image is composed of bytes and that altering those byte values in a smart way allows for message embedding or how we can use the file specifications to our advantage and hide information in the file metadata or after the offset where all renderer programs will stop parsing. All of the previous examples deal ony with the spatial domain of the format because they work on the raw bytes and consider each of them to be completely individual and self-sustaining entities that can be altered for steganographic purposes. However, there is another domain that works differently than the spatial one and it is called the frequency domain. In this domain the final representation of the stored data is done by combining the entire range of frequencies into the equivalent signal, usually by using a transform function, the most common being the Fourier transform. The advantage of the frequency domain is that it helps split a signal of any type into clear and distinct sinusoids that are easier to perform complex tasks on.

In figure 4.2 we can see the differences between the time domain and the frequency domain: the former evolves over time and is highly irregular in most cases(in real world there will never be a perfect sinusoidal signal and will be more similar to the last example), while the latter is easier to manipulate and identify, and through the aforementioned transform functions can be converted back into the time domain.

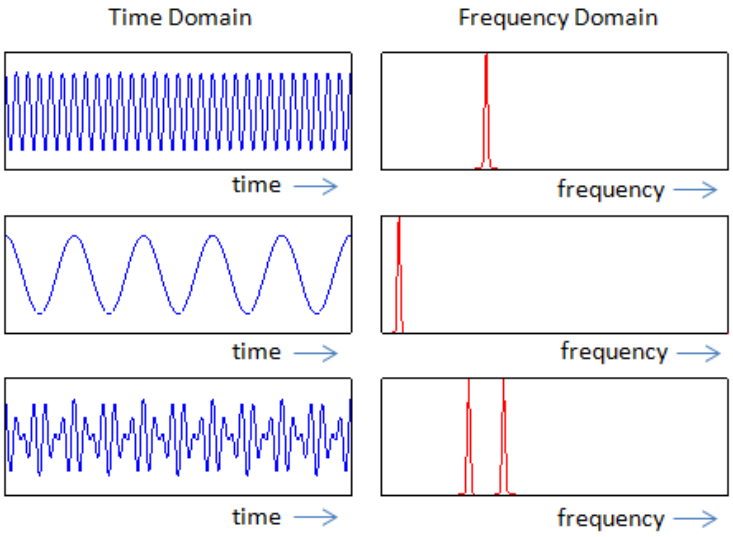


Figure 4.2: Time domain vs. frequency domain.

In the digital images world, the frequency domain is used to know by how much pixels variate from one another, in other words, the rate with which the pixels change. The most common format used for images that takes advantage of the frequency domain is JPEG, but since it is the only known format which uses sinusoids when rendering the image, the authors chose not to include it since the steganographic surface was somewhat limited. However, in the digital audio world every sound will eventually become an analog signal before reaching our ears so it is more common to see algorithms developed specifically for this format that involve altering the sinusoids to our advantage.

For example, we have the technique described in the article Frequency Domain Steganography by Ganier et al.[8] where they showcase the most basic way of embedding an audio file within another audio file: since both files have audio signals that are stored as frequencies, it is possible to "compress" the signals so that they only occupy a very specific frequency range and hide the message within the inaudible frequencies of the carrier, a much trivial task when not working in the time domain. This method takes advantage of the human physiology we mentioned earlier and achieves a high rate of success. Similar work has been done by Westfeld et al. [21] where they took the audio signal generated by the Slow Scan Television(SSTV) and embedded it into another carrier audio signal without any audible noise being generated that could alert intruders. On the more technical and software side of things there are applications such as Audacity that can integrate plugins written in a language called Nyquist that are specifically designed for frequency encoding secret messages, along with Matlab implementations of the aforementioned papers and many more.

Furthermore, there is also the option of steganography done within the spectrogram of a signal. A spectrogram is the visual represention of the entire spectrum of the frequencies as it evolves over time, basically getting the frequency domain and reintroducing the time axis into it. It is by far the most common place of hiding messages because it has been popularized by Capture the Flag competitions as entry level challengs and easter eggs in the video game community created by the developers. An example of hiding a key inside the spectrogram of an audio file can be seen in figure 4.3, made by the developers at DICE for their community in a secret challenge[2].

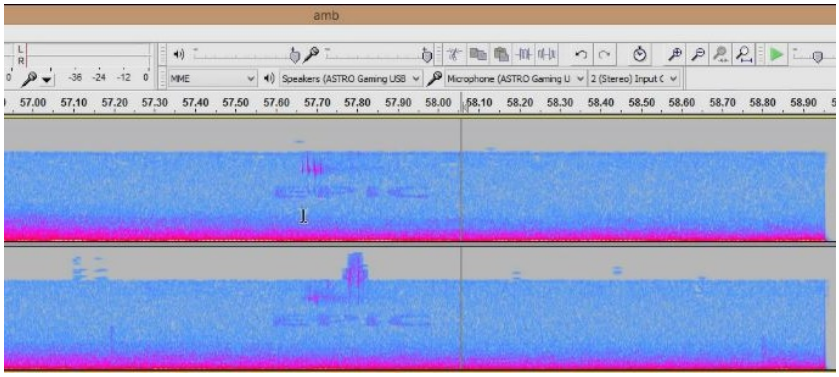


Figure 4.3: Message within the spectrogram viewed using Audacity.

4.3 Waveform Audio (WAV)

The Waveform Audio format commonly known as WAV is a popular file format for storing high quality digital audio files originally built by Microsoft. It bears many similarities to the PNG format in the internal structure/composition of the file: both begin with a very specific sequence of bytes (also known as the magic bytes) that help classification applications identify them, are separated into multiple parts (also known as chunks) that have their purpose and are extremely common in the modern day multimedia. WAV is one of the most common file formats that can be found in the digital world and that makes it a viable candidate for the message carrier role in the steganography domain. Like almost all other formats designed by Microsoft it has a few variations and extensions of the original specifier to accomodate more metadata or higher quality audio, but since they are much rarer than the simple and original standard, we will not focus on those altered versions in this thesis.

In the table below we can see the general structure of almost any WAV file that contains PCM data [1] which are basically the default option of every digital audio recording software. From the table we can notice how there aren't any metadata fields that are less important which could make for a good initial foothold in the steganography process.

Chunk field	Field Length	Description
Chunk ID	4 bytes	Always equal to "RIFF"
Chunk Size	4 bytes	Length in bytes of remaining file
Format	4 bytes	Always equal to "WAVE"
Subchunk ID	4 bytes	Always equal to "fmt "
Subchunk size	4 bytes	Always equal to 16
Audio Format	2 bytes	Equal to 0x0001
Nr. of channels	2 bytes	Channel count (1 mono, 2 stereo etc.)
Sample Rate	4 bytes	How many samples per second(6000Hz, 44100Hz, etc.)
Byte Rate	4 bytes	Product of SampleRate, number of channels and the byte per sample (BitsPerSample/8)
Block Align	2 bytes	Actual byte count for a full sample (both channel in a stereo file for example result in 4 bytes for a full sample)
Bits Per Sample	2 bytes	How many bits in a sample for a single channel
Subchunk ID	4 bytes	Equal to "data"
Subchunk size	4 bytes	Length of actual data
Data	*	The actual data

However, we see a great deal of advantages as well based on the file structure:

- **Uncompressed data** means that we do not have to go through the process of decompressing it, editing it, and then recompressing it in order to store any information. We are able to go sequentially through each byte and without being careless we can alter it in order to perform least significant bit insertion and hide the message, making WAV a good carrier for any type of sequential steganography. There are a few issues that could arise, usually samples are stored on two bytes and altering the least significant bit of each byte will alter the associated sample in a much much greater way, so we must keep in mind the bit depth of the samples when inserting the data. Furthermore, we must be careful with the audio channels of the file since modifying only one channel will possibly introduce noise to the audio stream, raising red flags to intruders and blowing the cover of the carrier.
- **The chunked structure** of the format leads to the ability to insert custom chunks with our data because most parsers will simply ignore them and look for the relevant chunks that they need. There are also a few types of chunks such as INFO or JUNK that could be used for a safer and more covert communication because they are a part of the standard and that means there will be no risk of crashing the parser if they find a chunk they could not identify.
- **Length is known beforehand** means that we know where the audio data ends just by parsing a few relevant metadata fields. This means that it is very easy to compute the offset where the file ends and we can begin embedding secrets while avoiding breaking the parsers or disturbing the audio data.

In conclusion, WAV is an impressive carrier format worthy of being used in the steganography process due to the fact that it is commonly seen on the Internet, has no compression that can affect the secret message and can easily work with all the algorithms presented over the course of this thesis.

4.4 The MPEG-1/2 Audio Layer III (MP3)

Moving Pictures Experts Group or MPEG for short has plenty of both official and unofficial standards, multiple versions etc. The focus of this subchapter will be on MPEG Versions 1 and 2 (the only officially

[1] PCM is an abbreviation for Pulse-code modulation which is the most common way of storing digitally the sampled analog audio signals.

accepted standards), or to be more precise, Layer III of these versions. As you can already see, there are tons of variations of what should be a single and standardized format, but all of them exist for a good reason and have their own use cases, even though they differ in available bit rates or the sampling rate frequencies range. Talking about all of them would be pointless and would easily take hundreds of pages to showcase as they have been the result of many years of research and evolution, so instead we are going to focus on the most common standard that is available in the digital world, which is even more popular than the aforementioned audio file format, and that is MPEG Version 1 Layer 3, hereby known simply as MP3. While technically any version of the MPEG that uses Layer 3 is formally known as MP3, versions 2 and 2.5 are much rarer since they offer smaller bitrates and smaller frequency sampling rates, making them inadequate for audio usage in the modern day.

The biggest reason MP3 has been one of the most popular audio file formats used in the world since its invention back in 1993 is due to its ability to replay high quality audio samples while still keeping a very small disk usage overhead. It takes advantage of the Huffman coding to reduce the total length of the file, making it able to retain the same perceptible audio quality to the human ear while still compressing the final size between 75% and 95% of other uncompressed formats, such as WAV[13]. However, due to the usage of compression algorithms, the long time the format was in development, the high number of standards and substandards and the inclusion of other formats in the same binary stream as the audio samples for various reasons, make the MP3 an incredibly complex format that need more advanced parsers than the ones found in other trivial formats, such as BMP or WAV.

Before beginning to talk about steganographic algorithms that could be applied to the format, we first need to understand the MP3 binary stream to make sure we do not alter bytes that could mark the file as corrupted, such as checksums or important headers. Nowadays, the structure of most MP3 files consists of two parts:

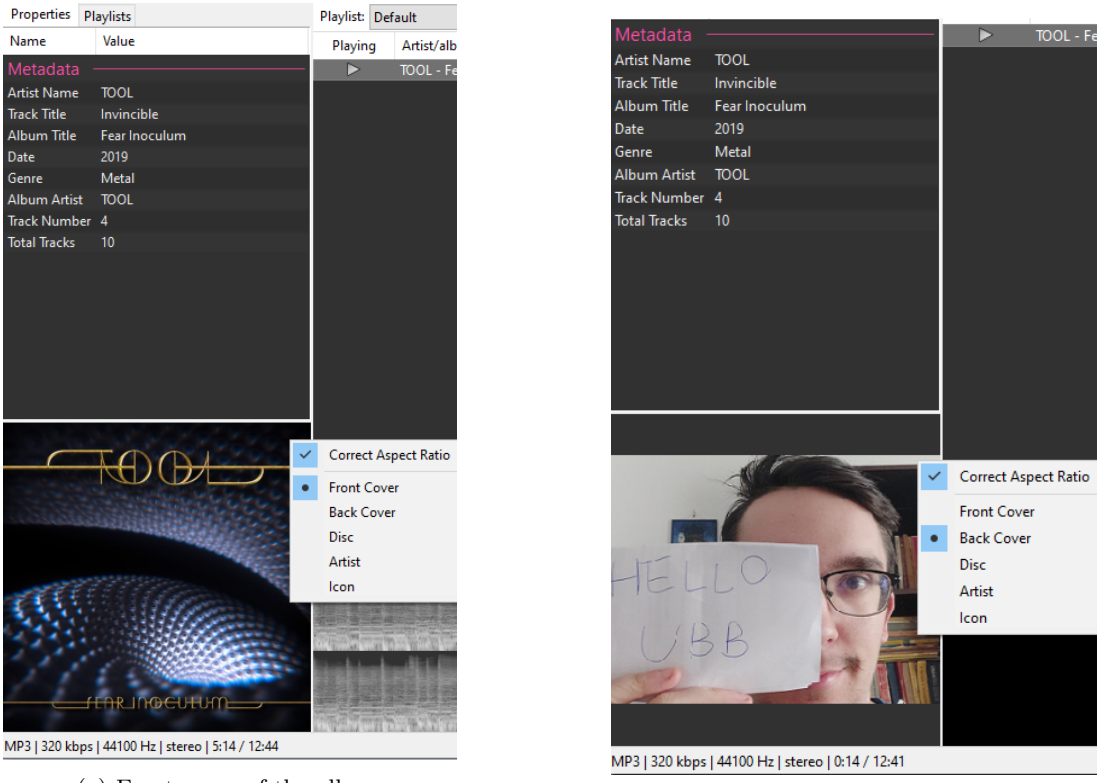
- The **ID3** part, the metadata container where information such as the track artist, album name or even album art is stored.
- The actual **MP3** audio data, contains the audio samples in a special format.

ID3 is a standard used to store file metadata that is most commonly seen alongside MP3 files, which is why it is important to discuss it because it can offer a few interesting options that can be viable secret message carriers. It is always found at the beginning of the audio file such that any parsers that are not interested in the metadata will need to read only a few bytes in order to get the offset to the actual sound data. Similar to the PNG, it is a chunked structure that may contain multiple chunks (or frames as the standard specifies). We can see in table below the general structure of the ID3v2 header(the most common type of header, the previous versions being flagged as obsolete):

Chunk type	**Chunk size**
Header (REQUIRED)	10 bytes
Extended Header (OPTIONAL)	Variable Length
Frames (at least 1)	Variable Length
Padding (OPTIONAL)	Variable Length
FOOTER (OPTIONAL)	10 bytes

We can see from the table the similarities to other chunked file formats. Since modifying any of the important bytes from the header/footer or padding can lead to corrupted and unparsable files, we shall stray away from the. However the frames are a perfect environment for our messages since we can have almost any number of them (limited by the 4 bytes or 28 bits that form a synchsafe integer representing the total size of the ID3 header), along with a high range of official frame types that we can pick from that most parsers will probably ignore. We can see in the figure below a frame with the ID equal to APIC (short for attached picture) which is used to usually embed images into the stream that represent album art or covers. Such an example can be seen in figure 4.4a

Our advantage is that there is no limit of frames with the ID equal to APIC there can be in a file so it is a prime target for metadata steganography when our message is a simple image. By changing a few bits we can mark the image as being the back cover of the song or we can even make it invalid such that it will exist in the binary stream but no renderers will show it. The process of parsing the metadata is simple after reading and understanding the documentation and results in a trivial encoding process. The difficult part is when attempting to write our image since it has to be inserted without breaking any of the other frames. After the entire process is done, we get the result in figure 4.4b.



(a) Front cover of the album Fear Inoculum by Tool

(b) Back cover containing our hidden image

Figure 4.4: Hiding images inside album art covers

Unfortunately, since MP3 is a much more complex format than we are accomodated with we will not look into the process of Least Significant Bit insertion as that would entitle too much trouble for such a simple result and since it is by default a compressed format it is not a significant loss because the available domain is smaller and we would not be able to achieve an acceptable performance. This is the reason why metadata and after-end steganography are the only viable methods when dealing with the MP3 file format.

In conclusion, MP3 is one of the best formats for metadata embedding since it has so many types of frames that are impossible to keep track of and that no user will possibly notice if they are not looking specifically for it.

Chapter 5

The Steganos Project

Steganos is the name of the application that implements the algorithms enumerated in the earlier chapters of this paper and is the direct result of the work done by the authors. In this chapter I will present what modern technologies and frameworks went into creating the application, how it is structured from an Object Oriented point of view, how fast it is and how I hope it will expand into the future.

5.1 Used technologies

5.1.1 C++

C++ is one of the lowest high-level computer programming languages. Designed by Bjarne Stroustrup, it first appeared in 1985 as a variation of one of the most popular languages at the time, C. It is one of the most efficient modern languages mainly because it has been designed with performance and flexibility in mind, just like its predecessor. C++ has gained a lot of traction from big companies like Intel and Microsoft which needed a programming language that could be used for operations ranging from basic kernel functions to highly specialized Object-Oriented projects with Graphical User Interfaces[17].

Since its conception, C++ has grew substantially by adding support for generic and functional features to ease the development processes and getting standardized by the International Organization for Standardization probably helped as well because it meant no more obscure variations of the language, allowing programmers to follow only one standard, ending up with even more portability and stability of the applications.

In the modern day, it is used almost everywhere: the kernel of various operating systems, the transaction software used by banks, drones and airplanes, embedded systems such as the Arduino or Raspberry Pi, and now in the Steganos Project as well under the latest approved standard of the language, C++17.

C++ inherited from C the file types: headers and sources. The header files (the files that have a .h or .hpp extension) are where the standard recommends to store the class declarations, function prototypes, constants declarations, no definitions should take place in a header file. There are a few exceptions to this rule, the most common one being a rule that states that all templated functions and functions are to be declared and defined in a header file otherwise it will not work properly when dealing with multiple instances of different types. In the case of the source files (the files that have the .c or .cpp extension) the standard suggests to only be used for the definition of the class methods or other general functions. Following the standard results most of the time in a clean and well defined separation in the code, allowing for high cohesion and low coupling in all projects built using C++.

The reason for choosing C++ as the language for the Steganos project is simple: it allows for the programmer to work on the raw bytes of files in an extremely easy way, reading and parsing them, bit-wise operations and writing to file, all the common low-level operations that are highly valued when working in the steganography field. But the high level aspects of the language also permit for dealing with complex tasks, such as modern cryptography algorithms applied to robust byte streams or holding all the information in different data structures.

5.1.2 CMake

CMake is a cross-platform open-source tool designed for managing the build process of software based on the C++ programming language. It has been created in the year 2000 and ever since then it stayed compiler-independent using simple configuration files that generate the adequate makefiles to be used in the users environment while building the target project[1].

CMake uses files called CMakeLists.txt that contain the commands to be used by the internals of the tool in the building process. It is required that one file is in the root of the project before running the CMake process, with the possibility of adding a .txt file in the subdirectories in order to indicate special cases that require a different approach.

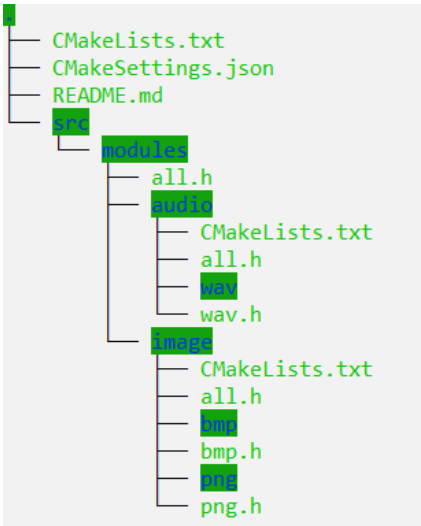


Figure 5.1: Folder structure of a project using CMake

Each command in CMake has the same format: COMMAND (args..). Using this format, users are able to build even the most complex software projects in the form of simple executables, dynamic or static linked libraries, etc. Steganos uses CMake because it is an extremely effective tool in the building process and it allows for separating the logic of the project into distinct modules i.e. the audio module, the image module, the general usage module, and linking them in the end into a simple executable. Most projects use only a small subset of the commands available in CMake, the most common being:

- **CMAKE_MINIMUM_REQUIRED** is used to mark the minimum version of CMake that is required to be installed on a system in order to build the project.
- **SET** is used to assign a value to a CMake variable that is used while building, similar to environment variables found on all operating systems.
- **PROJECT** for naming the project, important step in active Continous Integration and Continous Development environments.
- **INCLUDE_DIRECTORIES** for marking the directories containing the header files in order for the compiler to be able to link the source files and header files.
- **ADD_EXECUTABLE** for creating an executable after build the project from the source files given as arguments.
- **ADD_LIBRARY** for creating a library or module with the given source files.
- **TARGET_LINK_LIBRARIES** for linking (pre)defined modules or executables between eachother.

5.1.3 CXXOpts

CXXOpts is an open-source C++ library that is meant to be a lightweight command line option parser[11]. It is meant as the C++11 and beyond alternative to other libraries such as Commons CLI for Java or Argparse for Python. It was initially created by user jarro2783 on Github and nowadays is actively developed by the community using the Git commit and pull request system. CXXOpts is made to replicate the handling of the help argument and the merging of multiple parameters commonly found in *NIX command line binaries, without using any additional dependencies in a header-only file.

5.1.4 Lodepng

Lodepng is an open source C++ library developed by Lode Vandevenne used for image processing for pictures stored under the PNG format [19]. It is very useful for decoding the image data before beginning the steganography process, making it easy to modify the data and to encode it back into a PNG file that follows the standard format. Lodepng works on every version of C++, contains only two files(a source file and a header file) and is rated to be one of the fastest libraries for PNG image processing. Steganos uses Lodepng to parse PNG files and to obtain the image data byte stream from the zlib compressed stream in order to be able to work with the implemented steganographic algorithms.

5.1.5 Robot36 SSTV Engine

Robot36 is a library created by Ahmet Inan in order to be able to encode and decode images using the Slow Scan Television (also known as SSTV) format, more specifically the Robot36 standard. The library is open-source and is available on github[9]. By default it supports only color BMP images that are 320 pixels wide and 240 pixels tall and can be compiled using any C compilers only under a Linux system which has the Advanced Linux Sound Architecture (ALSA) libraries installed. Since Steganos is a C++ project that the authors intended to be cross-platform, we have forked the original project and refactorized the code to be buildable and to work on the Windows architecture as well. After some more refactorization is done, we have every intention to release our version of the SSTV Engine and to create a pull request to the original repository.

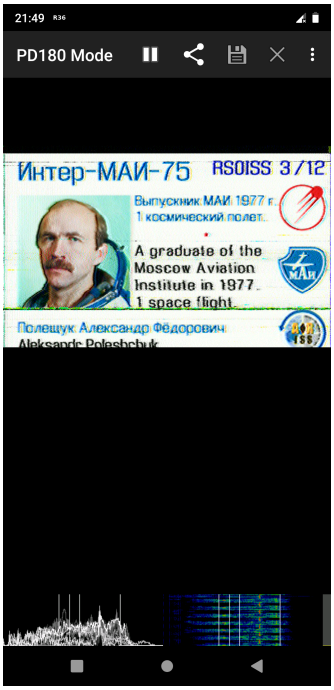


Figure 5.2: SSTV Engine decoding transmission from the International Space Station

5.1.6 Jenkins

Jenkins is one of the biggest open source automation servers that is meant to automate the process of building, deploying, testing etc. any project. Developed in Java, it supports a big range of popular languages and frameworks, either by default configuration or by installing a plugin available on their developer page. We picked Jenkins for Steganos due to previous working experience of the authors with it and since it has the ability to build and release C++ and CMake projects using a single plugin it has proven to be an extremely valuable asset in maintaning a continuous integration and continuous delivery schedule of the project.

Figure 5.3: Jenkins management interface of the Steganos job

5.2 Application architecture

The Steganos Project is an application that uses the Object Oriented Programming aspects in order to achieve the best performance while maintaining high cohesion and low coupling between the components of the codebase. The project is structured in multiple modules[1] that were built to be as independent as possible with the goal to provide steganography functions for the supported file formats. The main module contains the source files for the entry point of the application which processes the arguments given via the command line along with the code for some of the utilities, helper function that help convert pixels between different representations(YUV, RGB, BGR, YCbCr) or bit-wise operations such as toggling the last significant bit or building a byte using the LSBs from a byte stream. The algorithm module declares and defines the algorithms that are implemented in the application, all of which have been discussed thoroughly in the earlier chapters of the thesis (LSB insertion, embedding secrets into file metadata, use-after-end encoding). The remaining modules all follow the same design patterns and are responsible for the parsing of the cover files and the encoding or decoding of hidden digital files inside them.

[1]To be technically correct they are only different folders because the latest available C++ standard as of the time of writing this thesis is C++17 which does not understand yet the concept of modules. However developers should rejoice over the announcement made by Microsoft to standardize C++ modules starting with the C++20 standard[10].

We can see in figure 5.4 the overview of the entire application and what each module contains more specifically. This is only a very simplistic rendering of the module deployment, we will delve in for a more in-depth perspective later this chapter.

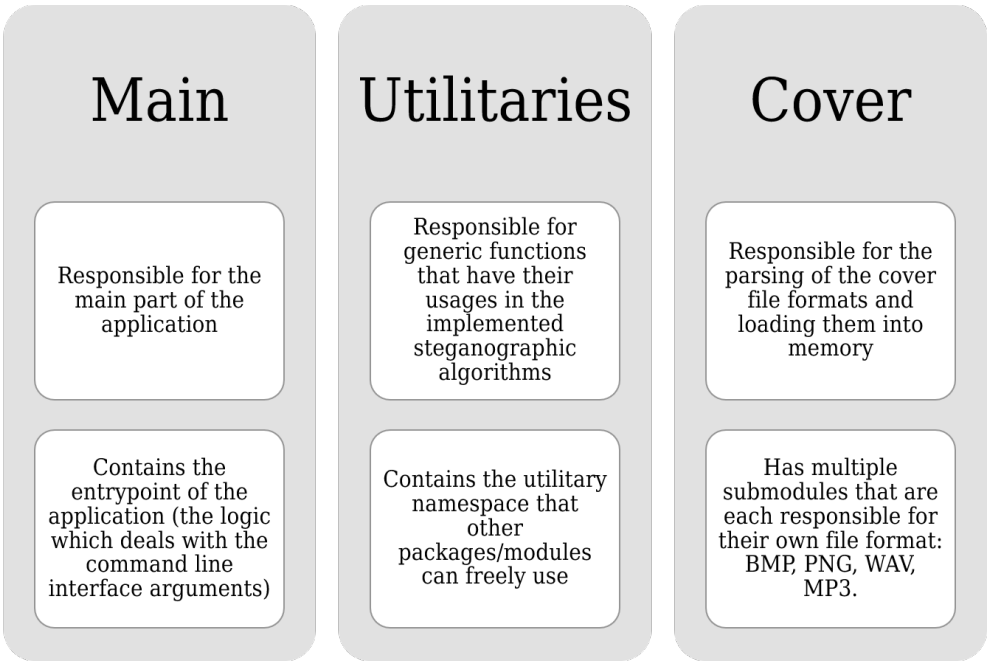


Figure 5.4: Overview of the Steganos modules

In figure 5.5 we can see a class diagram of the general struture of each of the four aforementioned cover submodules (the codebase which deals with parsing the cover files that allow for the steganography algorithms to have the necessary data to work with).

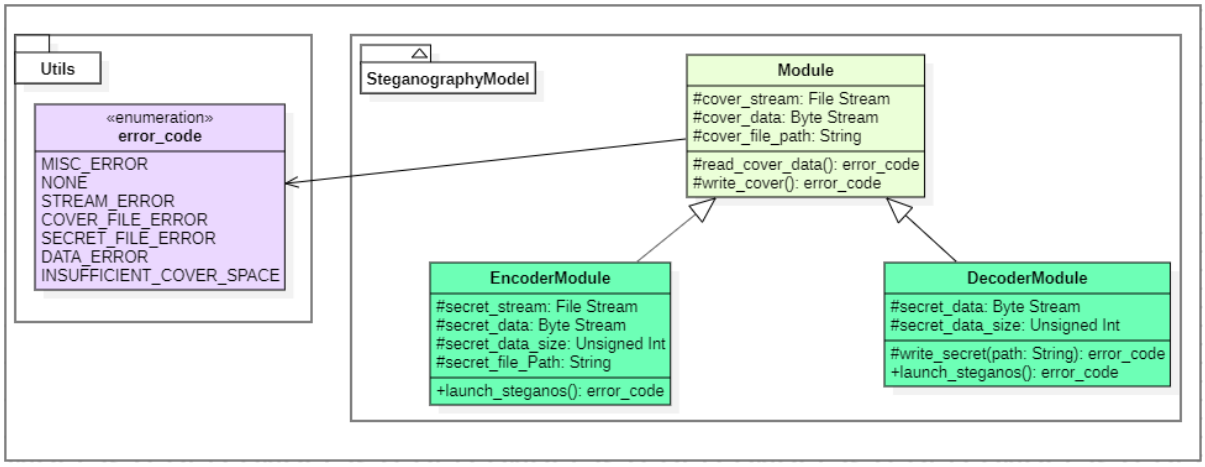


Figure 5.5: Structure of a cover module

After seeing how the project is structured into modules and submodules, what classes, interfaces, methods and attributes we have, it is also very important to understand how the data flows from the moment we launch Steganos to the moment the output file is generated (the output file is either the cover file containing a secret message, either the actual secret message, depending on the option used). In figure 5.6 we can see a flowchart of the application containing the key points of the entire process.

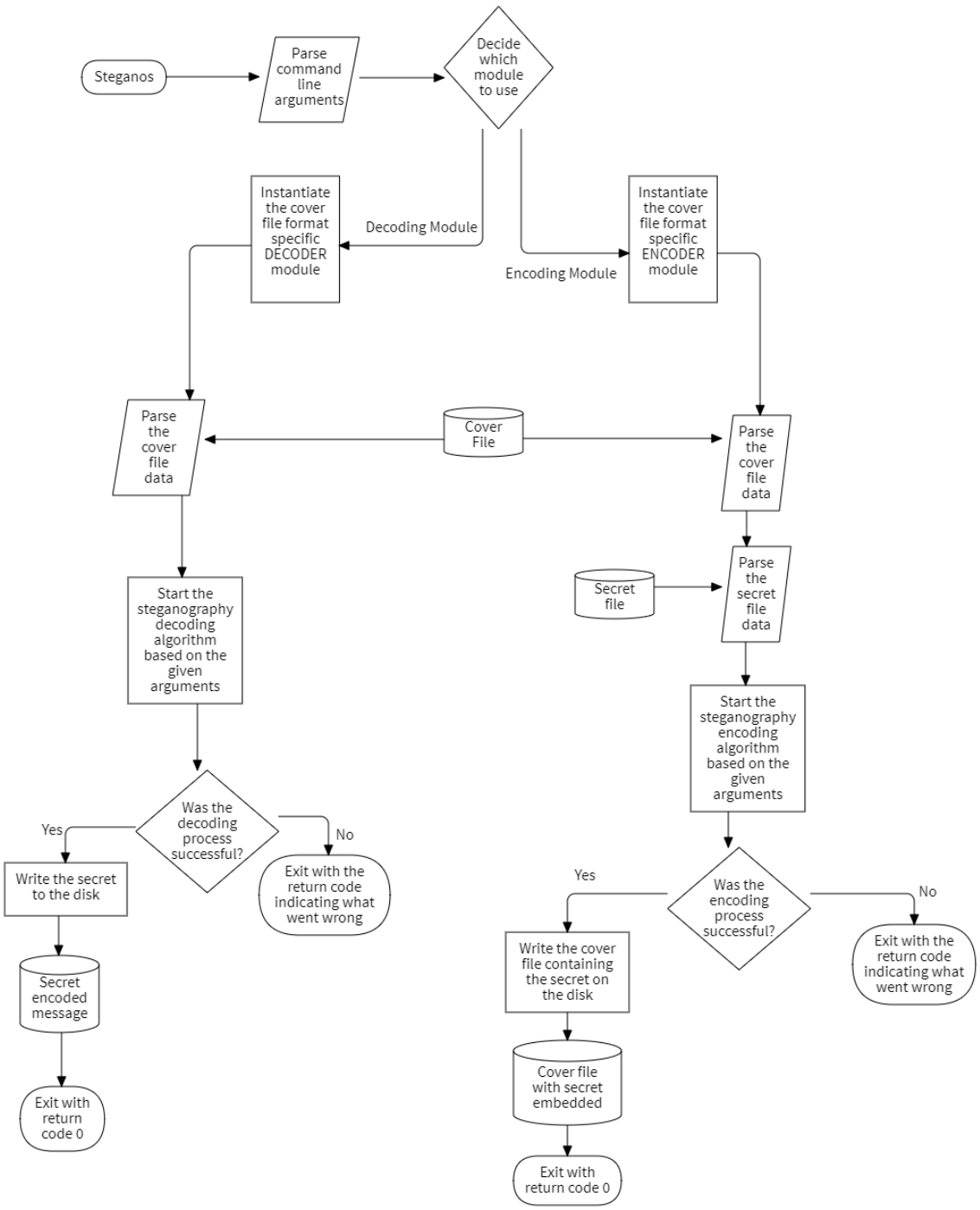


Figure 5.6: Steganos code flowchart

To make sure that the project will always be in at least a buildable state after any commit is done we are using Jenkins. It is very important to make sure that the code compiles and that if there are any errors, the authors are aware of them and will have the option to hold back the release until the issues are fixed. On a more related note to the Steganos application, we have the following job configuration for testing each new version before releasing:

- Maximum number of build to keep is set to 5 to conserve total storage used.

- Each new commit on the master branch will activate the hook trigger to let Jenkins know that a new version is available and that it should try to build it.
- Using the "Unix Makefiles" generator to set up the CMake build files with the build type set to "Release".
- Publish the build result to the project page to keep it up to date to the current release or trigger custom personal alerts to notify in case of failure.

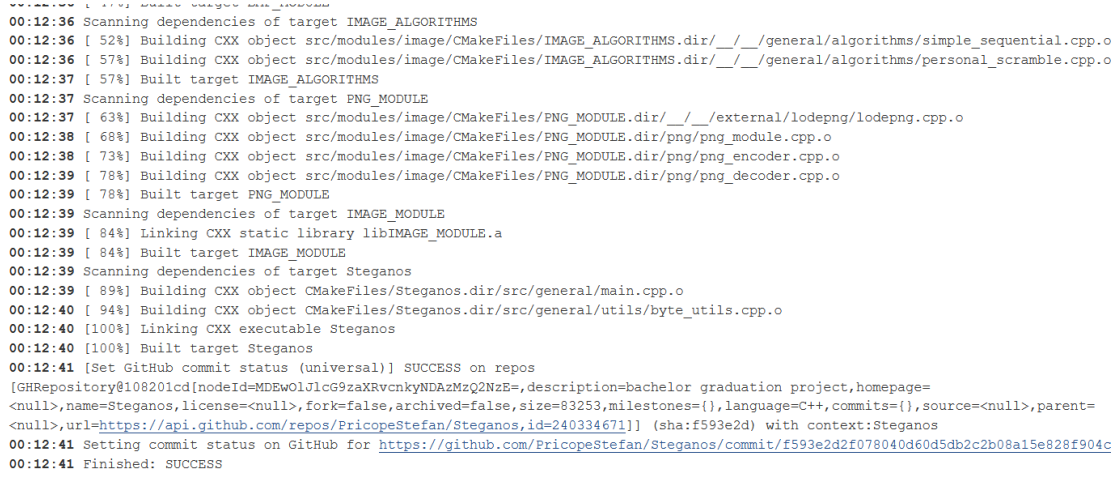


Figure 5.7: Jenkins log of the Steganos job build

5.3 Implementation details

Steganos contains many critical parts that are required to succeed in order to work properly. If any part of code or function call were to fail we would need to know so that we can properly interrupt the process on our terms, free any dynamically allocated memory, log the error messages so that developers are aware of any bugs that may appear or if the application is in a release form, notify the client of what has happened and why the application failed. Since C/C++ is a rather monstrous amalgamation when talking about exceptions raised during runtime execution (at least in the standard libraries, more complex frameworks like Boost do not suffer as much from this), we have preferred to validate the function calls based on a more simple method, the code returned by the function which is basically a glorified integer that has some meaning attached to its values. You can see in listing 5.1 the exact define macro used in the project and in listing 5.2 an example from one of the encoder module constructors.

Listing 5.1: The TRY macro used for any critical operation

```
#define TRY(X) generic_assert((X), __FILE__, __LINE__);
inline void generic_assert(error_code code, const char* file, int line) {
  if (code != error_code::NONE) {
    printf("[Assert error]Message = %d in file %s at line %d\n", code, file, line);
    exit(static_cast<int>(code));
  }
}
```

Listing 5.2: Usage example of the TRY macro

```
WAVEncoderModule::WAVEncoderModule(const char* cover_file_path, const char* secret_file_path) :
    WAVModule(cover_file_path)
{
  TRY(utils::load_stream(secret_file_path, secret_stream));
  TRY(utils::read_byte_stream(secret_stream, secret_data, secret_data_size));
}
```

Now that we know the macro that used to ensure the success of the critical operations, we can begin looking at some of those functions that can't fail. The first functions that we will be looking into will be those that perform Input-Output (shortened to IO) operations, such as reading or writing the raw bytes of a file. We can see in listing 5.3 how we open the stream to the file and then in listing 5.4 how the function that actually reads the contents of the file looks like. It is important to have as many checks as possible to ensure that any issues that may arise are dealt with and the execution of the program is not interrupted.

Listing 5.3: The TRY macro used for any critical operation

```
error_code utils::load_stream(const char* file_path, std::ifstream*& stream, std::ios_base::openmode
    stream_options) {
   if (stream != nullptr) {
      //stream is already open
      return error_code::STREAM_ERROR;
   }
   stream = new std::ifstream(file_path, stream_options);
   if (stream->is_open()) {
      return error_code::NONE;
   }
   //the opening of the stream failed, deleting it
   delete stream;
   stream = nullptr;

   return error_code::STREAM_ERROR;
}
```

Listing 5.4: The TRY macro used for any critical operation

```
error_code utils::read_byte_stream(std::ifstream* stream, uint8_t*& byte_stream, uint32_t& stream_size) {
   stream->seekg(0, std::ios::end);
   std::streamsize file_size = stream->tellg();
   stream->seekg(0, std::ios::beg);

   byte_stream = new uint8_t[file_size + 4];
   if (stream->read(reinterpret_cast<char*>(byte_stream + 4), file_size)) {
      stream_size = (uint32_t)(file_size + 4);
      //adding the stream size info to the byte stream as the first 4 bytes
      byte_stream[0] = ((uint8_t*)&stream_size)[0];
      byte_stream[1] = ((uint8_t*)&stream_size)[1];
      byte_stream[2] = ((uint8_t*)&stream_size)[2];
      byte_stream[3] = ((uint8_t*)&stream_size)[3];
      //succesfully read all the secret file bytes
      return error_code::NONE;
   }

   delete byte_stream;
   stream_size = 0;
   return error_code::STREAM_ERROR;
}
```

After we are done with the IO outputs, it is time to actually perform some reads as well. One of the first steps is to parse the metadata which is generally available as the first bytes of the file. However, there are 2 major issues that may arise: that structure is not respected, either because the file is using an extended version of the standard that adds other fields or simply because the file is corrupted, and secondly, the compiler is actively working against us because we are not doing something most programmers do, like misaligning the stack. The second problem usually happens when trying to read multiple bytes at once and then trying to map that memory to a structure. In listings 5.5 and 5.6 we see how to declare those structs such that any of the aforementioned problems are not an issue, and in listing 5.7 we see how exactly the reading and parsing is done in the code.

Listing 5.5: BMP header structure

```
#pragma pack(push, 1)
struct BMPMetaStruct {
   //generic file header
   uint16_t signature;
   uint32_t file_size;
   uint32_t reserved;
   uint32_t data_offset;
   //image info header
   uint32_t size;
   uint32_t width;
   uint32_t height;
   uint16_t planes;
   uint16_t bits_per_pixel;
   uint32_t compression;
   uint32_t image_size;
   uint32_t x_pixels_per_meter;
   uint32_t y_pixels_per_meter;
   uint32_t colors_used;
   uint32_t important_colors;
};
#pragma pack(pop)
```

Listing 5.6: WAV metadata struct

```
#pragma pack(push, 1)
struct WAVMetaStruct {
   uint32_t chunk_id;
   uint32_t chunk_size;
   uint32_t format;
   uint32_t subchunk_1_id;
   uint32_t subchunk_1_size;
   uint16_t audio_format;
   uint16_t num_channels;
   uint32_t sample_rate;
   uint32_t byte_rate;
   uint16_t block_align;
   uint16_t bits_per_sample;
};
#pragma pack(pop)
```

Listing 5.7: Reading file data directly into custom structure

```
//declaring the variables
std::ifstream* wav_stream = nullptr;
WAVMetaStruct cover_metadata;
...
//opening the stream
TRY(utils::load_stream(wav_file_path, wav_stream));
...
//actually reading the metadata into the struct
wav_stream->read((char*)&cover_metadata, sizeof(WAVMetaStruct));
```

All of the code presented will eventually have to go through the compilation process. We mentioned in the used technologies section that we are using CMake specifically for this purpose. Using few lines of code can help us build static or shared libraries, interfaces, objects and executables. This is what the code below does. It is the root level CMakeLists.txt that builds the main executable of the project, adds the subdirectories containing other CMakeLists.txt files to build the other modules as well, and links the needed libraries together in order to make the final executable work.

```
cmake_minimum_required (VERSION 3.8)
project ("Steganos")
include_directories(${PROJECT_SOURCE_DIR}/src)

add_executable (
        Steganos
        ${PROJECT_SOURCE_DIR}/src/general/main.cpp
        ${PROJECT_SOURCE_DIR}/src/general/utils.h
        ${PROJECT_SOURCE_DIR}/src/general/utils/byte_utils.cpp
        ${PROJECT_SOURCE_DIR}/src/general/utils/cli_utils.cpp
)

add_subdirectory(src/modules/image)
add_subdirectory(src/modules/audio)

target_link_libraries(Steganos IMAGE_MODULE AUDIO_MODULE)
if (WIN32)
        target_link_libraries(Steganos wsock32 ws2_32)
endif()
if (UNIX)
    target_link_libraries(Steganos stdc++fs)
endif (UNIX)

target_compile_features(Steganos PRIVATE cxx_std_17)
```

5.4 Using the application

Steganos was designed to be a very simple command line tool that is fast, reliable and portable. Using Cxxopts we have tried to implement an intuitive way of passing the arguments to the program, such that they all make sense and that any of the available configurations and command executions are easy to understand. You can see from the program output below that it is very easy to learn the options available to the end user by adding the -h flag and see what is supported and what not.

```
D:\Projects\Steganos>Steganos.exe
You must specify one cover file.
Do Steganos.exe -h for more information.
D:\Projects\Steganos>Steganos.exe -h
Simple steganography project created as a part of my bachelor's thesis
Usage:
  Steganos [OPTION...]

 Encoding/Decoding options:
  -c, --cover FILE   The file that serves as a cover for the secret message
  -s, --secret FILE  The secret file which will be embedded into the cover
                     file - ENCODING ONLY
 General options:
  -o, --output FILE             Name of the output file (default: output)
  -v, --verbose                 Verbose output
  -m, --method METHOD           The method to be used when encoding/decoding
                                the message
  -p, --passkey PASSKEY         The password used to secure the message or to
                                decipher it (default: password)
  -h, --help                    Prints this help message
```

Based on the information above, it is almost trivial to come up with working command examples based on the help menu we displayed earlier. We can begin looking at some examples to see how Steganos works. Let's assume that we want to encode a secret message stored in a txt file within a BMP file. All we would need to type is the path to the secret file and the path to the cover file. The output can be seen below.

```
D:\Projects\Steganos>Steganos.exe --cover marbles.bmp --secret secret.txt
Done
```

That's all there is to it. However to some the need to type so many characters without any autocomplete (because option autocomplete is not commonly available in most shells) could be an annoyance so Steganos also offers the option for shortened option names to make the process quicker.

```
D:\Projects\Steganos>Steganos.exe -c marbles.bmp -s secret.txt
Done
```

We also have the option to enable verbose output to see debugging information in real time during the Steganos process and see exactly what it is doing. Furthermore, we can specify the name of the output file to be whatever we would like it to be.

```
D:\Projects\Steganos>Steganos.exe -c marbles.bmp -s secret.txt  -o secret_marbles.bmp -v
BMP Image data is found beginning at 54
Ignoring 0 bytes to get to the actual image data
Available methods:
SEQUENTIAL (default)
PERSONAL_SCRAMBLE
No encoding method specified! Picking default option.
Deleted image data and its associated stream
Done
```

If the user is confused as to what options are available for a given file/file combo during either of the embedding or the decoding processes, he can always add the -h flag to get the valid accepted methods.

```
D:\Projects\Steganos>Steganos.exe -c marbles.bmp -s sstv_me.ppm -h
Available steganography methods when using BMP files as cover:
SEQUENTIAL (DEFAULT)
PERSONAL_SCRAMBLE
Special option because secret file is .ppm : SSTV

Example program usages:
Encoding(must specify both cover and secret file):
         ./Steganos -c cover.bmp -s secret_message.txt -m personal_scramble
Decoding(must specify only cover file):
         ./Steganos -c suspicious_looking.bmp -o what_were_you_hiding.txt
```

5.5 Further work

Regarding the future of Steganos, I hope that it will be fairly fruitful and worthwhile. The entirety of this thesis we have only laid down the groundwork of the project and have talked about the most basic and essential part of the application. However, the possibilities for extending Steganos are almost endless and during the time while building the application and writing this thesis I have saved some of the more note-worthy ideas that came to my mind (in no particular order):

- **Additional file formats support**. Right now Steganos only supports 4 major file formats, two digital image formats (BMP and PNG) and two audio formats (WAV and MP3). However, there are a lot more file formats that are commonly used on the Internet and that could be viable picks as a cover in the steganography process. Some of those formats include but are not limited to: Joint Photographic Experts Group (JPEG images) format, Free Lossless Audio Codec files (high-quality audio files, commonly found on album disks), Graphic Interchange Format (GIFs) and many more.

- **Proxy server and chat application**. This is one of best development Steganos could get in my opinion. As mentioned earlier, right now the project is similar to only a core-engine library that just takes some input, processes it and turns it into an output. It is not integrated anywhere, it's just a simple CLI application. However, it would be nice to see it integrated into a chat application, following a very simple idea: run Steganos as a server process which accepts any kind of message and returns a multimedia file that has the secret embedded within and then send that image/audio/video file to the peers. They would be able to decode the message based on an already negociated process (the negociation took over a secure channel that any intruders would not have access to) and they would be able to reply in the same manner. The clients could also change the encoding/decoding process automatically and then notify the peers by also embedding the renegociation in the cover file. This application has the advantage that it only needs a brief secure environment for the initial connection and then be able to securely coomunicate over any type of channel. It would certainly be one of the most interesting usages for a steganography project, mostly because it would be practical enough to be used in real world situations and not just Capture the Flag challenges.

- **Windows integration for Jenkins**. At the moment of writing this, there is only one deployed instance of Jenkins running the Steganos job and it is based on Linux. That means that the only types of files that are automatically generated and tested are only the Linux binaries since there is no easy way of also building the Windows binaries. Some research has revealed that there is the option of adding a Windows based Jenkins instance in a master-slave configuration that would technically allow the full building and testing of the executables but we have not yet managed to implement this.

- **Additional algorithms**. Most algorithms described in this paper are implemented in the Steganos project but there are plenty viable ones (especially those that rely on scrambling the data) that have not seen yet the light of day. It would certainly be interesting to add them and compare them to existent implementations or to the other methods.

- **Encrypting the secret**. Even though the option is specified in the help menu, it is not yet possible to encrypt the actual secret, mainly because I could not decide on which encryption methods to make available and because I also would rather focus on building the core codebase and to implement the steganography algorithms first. However, it is almost mandatory that the option of encrypting the secret before the embedding process is added in a future release to increase the security of the contents.

- **Validating the decoding result**. Right now Steganos only spits out the binary stream that it finds embedded within the cover file but does not perform any kinds of validation to see if it actually extracted the right secret file or not. There are multiple fixes for this problem, however in my opinion the best solution is to also include a hash of the file in the cover as well (alongside the length of the secret file) and to check after extraction if the embedded hash and the hash of the result are equal. It may sound trivial but in reality it could be difficult to implement due to some constraints caused by C++ and due to the fact that it also reduces the total space available for hiding.

Chapter 6

Conclusion

In conclusion, I believe that steganography in modern multimedia is in a really interesting position: it is not advanced enough to win over cryptography and become the main pillar behind secure communications over unreliable networks and most likely this will never change since steganography by definition will always be slower because it relies on a carrier for securing the message instead of just securing the message itself. However is steganography still an extremely important method of protecting information such that only the entrusted parties that are aware of the used algorithms will be able to extract the hidden message. It is my belief that there are countless more steganographic innovations to be made in this field that researchers will discover in the future, expanding the number of applications that steganography has right now beyond digital watermarking or Capture The Flag challenges in hacking competitions.

Taking a step back to look at the entire known history of steganography methods we can clearly see that it has evolved a lot: from the Roman Empire hiding messages on the shaved heads of their slaves, waiting for their hair to grow back and then send them away with important messages, to the modern day digital steganography that contains several impressive and complex algorithms in order to ensure high capacity and fast encoding/decoding processes. Given this evolution it is same to assume that more and more complex and well researched algorithms will appear in the future, bringing steganography closer to the spotlight of various information security topics and conferences.

Bibliography

[1] Cmake documentation. `https://cmake.org/documentation/`.

[2] Phantom program. `https://battlefield.fandom.com/wiki/Phantom_Program`.

[3] Ahsan, K., and Kundur, D. Practical data hiding in tcp/ip. In *Proc. Workshop on Multimedia Security at ACM Multimedia* (2002), vol. 2, pp. 1–8.

[4] Al-Bahadili, H. A secure block permutation image steganography algorithm. *International Journal on Cryptography and Information Security 3* (09 2013), 11–122.

[5] Al-Othmani, A., Manaf, A., and Zeki, A. A survey on steganography techniques in real time audio signals and evaluation. *International Journal of Computer Science Issues 9* (01 2012).

[6] Bellovin, S. Rfc3514: The security flag in the ipv4 header. Internet Requests for Comments, 4 2003.

[7] Deutsch, P. Deflate compressed data format specification version 1.3. RFC 1951, RFC Editor, https://www.rfc-editor.org/rfc/rfc1951.txt, 5 1996.

[8] Ganier, C. J., Hollman, R., Rosser, J., and Swanson, E. Frequency domain steganography. `https://www.clear.rice.edu/elec301/Projects01/smokey_steg/freq.html`.

[9] Inan, A. xdsopl/robot36. `https://github.com/xdsopl/robot36/`.

[10] Working draft, extensions to c++ for modules. Standard, International Standardization Organization, https://www.open-std-org/, Jan. 2018.

[11] jarro2783. jarro2783/cxxopts. `https://github.com/jarro2783/cxxopts/`, Feb 2020.

[12] Johnson, N. F., and Jajodia, S. Exploring steganography: Seeing the unseen. *Computer 31*, 2 (1998), 26–34.

[13] Musmann, H. G. Genesis of the mp3 audio coding standard. *IEEE Transactions on Consumer Electronics 52*, 3 (2006), 1043–1049.

[14] Petitcolas, F. A., Anderson, R. J., and Kuhn, M. G. Information hiding-a survey. *Proceedings of the IEEE 87*, 7 (1999), 1062–1078.

[15] Provos, N., and Honeyman, P. Hide and seek: An introduction to steganography. *IEEE security & privacy 1*, 3 (2003), 32–44.

[16] Shannon, C. Communication in the presence of noise. *Proceedings of the IRE 37*, 1 (jan 1949), 10–21.

[17] Stroustrup, B. *The C programming language*. Addison-Wesley, 2018.

[18] T. Boutell, e. a. Png (portable network graphics) specification. RFC 2083, RFC Editor, https://www.rfc-editor.org/rfc/rfc2083.txt, 3 1997.

[19] Vandevenne, L. lvandeve/lodepng. `https://github.com/lvandeve/lodepng`, Mar 2020.

[20] Webster, M. Metadata. `https://www.merriam-webster.com/dictionary/metadata`.

[21] Westfeld, A. Steganography for radio amateurs— a dsss based approach for slow scan television. vol. 4437, pp. 201–215.

[22] Westfeld, A., and Pfitzmann, A. Attacks on steganographic systems. In *International workshop on information hiding* (1999), Springer, pp. 61–76.