Video Steganography

COM3600 Research Project

This report is submitted in partial fulfillment of the requirement for the degree of Master of Software Engineering in Computer Science by James Ridgway

Author:
James Ridgway

Supervisor:
Dr. Mike Stannett

3rd December 2012

Signed Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used (where possible) with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: James Ridgway

Signature: _____

Date: 3rd December 2012

Abstract

Digital steganography typically centres on hiding messages in digital files. Typically, steganography focuses on hiding content in image and audio files. In comparison, the research and usage of multimedia files as covert objects remains limited. The aim of this project is to explore different methods for encoding messages in a multimedia container, utilising both the audio and video stream, and using steganalysis to determine the effectiveness.

Preface

From the outset I have documented a large proportion of my work and progress online via `http://www.steganosaur.us`, therefore some of the material outlined in this document may already have been published online.

In addition to documenting the progress of this project, I have developed several tools which are available online. Some of these tools were produced as part of our preliminary research into fundamental steganographic techniques. Some information relating to neighbouring fields, such as cryptography, can also be found on the website.

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

Steganography is an unusual aspect of security that is not commonly known, despite having a history that dates back thousands of years [Col03]. The roots of steganography date back as far as the Ancient Greeks, who provide us with the first description of a technique, which 1500 years later, was labeled "Steganography". The term is derived from two Greek words: *stegano* and *graphia*, meaning "covered" and "writing" respectively [Fri10, Col03]. Put simply, steganography is the practice of concealed communication where the presence of a message is secret.

Despite the Greek origin, the word "Steganography" does not appear in the literature until the 15th Century, when Johannes Trithemius uses the word in trilogy published in Frankfurt in 1606. The first two volumes, *Polygraphia* and *Steganographia* specifically discuss cryptography and steganography [Fri10, Col03].

1.1 The History of Steganography

1.1.1 First Evidence of Steganography

Whilst the term "Steganography" is only a few hundred years old, the concept of hiding and concealing messages has existed for thousands of years.

The earliest known written account of steganography being used is told by Herodotus (484-425 BC) [Her96], who tells how his master, Histiaeus, sent a slave to the Ionian city of Miletus with a message concealed on his body. The slave's head was shaved and the message was tattooed on his scalp. Once his hair had grown back concealing the message he was sent on his way to the city's regent, Aristagoras. Upon his arrival, the slave's head was shaved revealing a message persuading Aristagoras to revolt against the Persian king.

Herodotus also documents how Demeratus used a wax tablet to send a concealed message to Sparta, warning of the planned invasion of Greece by the Persian Great King, Xerxes. Demeratus removed the wax from the tablet and inscribed his message on the wood beneath before applying a fresh coat of wax. The tablet could then be carried and used normally. The hidden message was only revealed by scraping away all of the wax. Aeneas the Tactician, another Greek writer well-known for his various steganographic approaches and techniques, also proposed methods for concealing information in women's earrings, or using pigeons to deliver secret messages [Tac90].

1.1.2 Linguistic Steganography

Linguistic steganography is possibly one of the oldest forms of steganography. Aeneas the Tactician, again, described many linguistic techniques, which are now considered fundamentals of linguistic steganography. For instance, he describes altering the height of letters or marking particular letters with dots or small holes. Linguistic steganography has been used prolifically throughout history, and modern day variants of these techniques still exist today. Giovanni Boccaccio, a 14th Century poet, encoded over 1500 letters taken from three sonnets, into his acrostic poem, *Amorosa Visione* [Fri10]. This is possibly one of the largest examples of linguistic steganography.

Possibly the most interesting linguistic technique was proposed by Francis Bacon. Bacon's method allows messages to be encoded using a binary representation, by using normal or italic

font [Bac40]. The scheme proposed by Bacon is a precursor to modern steganographic techniques.

In 1857, Brewster proposed a photographic technique that would allow text to be shrunk down to a dirt-sized speck. Only under very high levels of magnification would it be possible to read the message. In World War I, the Germans used this technique to conceal large messages in the corner of post cards. The "microdot" technique used by the Germans was capable of hiding entire pages of text and even photographs, making them a powerful container of covert information [Fri10, JDJ03].

During World War II the following message was sent by a German spy [Kah67]:

> *Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on by products, ejecting suets and vegetable oils.*

By taking the second letter of each word the following message is revealed:

> *Pershing sails from NY June 1*

The type of linguistic steganography used above is sometimes called a *null cipher* [Rab04].

1.2 Modern Steganography

With the advent of computers and modern technology steganography is becoming more and more widespread. Image, audio and video files present interesting digital file formats for concealing information. The invention of the internet has provided a greater means of sharing information, and as such, it has become commonplace to share digital media. Media files are generally large in size, which has facilitated the hiding of large quantities of data.

1.2.1 Prisoners' Problem

Undetectability is an imperative component of steganography. Simmons famously illustrates this crucial fundamental through his description of the principle of the Prisoners' Problem [Sim83].

Alice and Bob are imprisoned. Alice and Bob can communicate with each other, but all of their communication is constantly monitored by warden Wendy. Alice and Bob want to hatch an escape plan, but if warden Wendy catches them trying to communicate secretly, they will be placed into solitary confinement. Alice and Bob decided to use steganography to communicate covertly. Alice and Bob must make sure that their communication is undetectable, since the mere presence of a secret message will alert Wendy and result in her placing both of them into solitary confinement.

1.2.2 Steganography, Security and Cryptography

Cryptography and steganography are often regarded as similar practices, and whilst both fields deal with secure communication, the differences between these two fields are plentiful.

For cryptography to be secure, a communication must be unintelligible – cryptography is only broken once the original message is understood. Thus, an encrypted message is secure if it cannot be read.

Steganography, on the other hand, is only secure if the existence of the message is not known. Once a covert message is revealed, steganography has failed, because something that was intended to be covert, has become overt.

Whilst these fields differ in their definitions of "secure", there are areas which are common to both domains. Kerckhoff's principle, which is native to cryptosystems, but is also applicable to steganography, states that a cryptosystem should remain secure even if everything relating to it is public knowledge - security should reside in the key [Kah67, Sch96]. This applies to steganography (and the Prisoners' Problem) - even if the steganographic algorithm is public knowledge, the existence of a message should remain unknown without the correct steganographic key.

1.2.3 Watermarking

Watermarking is a technique for hiding supplementary information in a file [Fri10]. This technique is similar to steganography, however there are some key differences. With steganography the data embedded should be covert and undetectable; in contrast it does not matter if watermarked information is easy to detect, the important factor is that it should be difficult to remove. Removing a watermark should result in significant degradation of the quality of the container file [Col03]. Watermarking is commonly used to help trace the origin of files. MP3 files purchased over the internet are regularly encoded with the details of the buyer and seller. The traceability and public knowledge of watermarking acts as a strong deterrent against online piracy. Watermarking is successful because of the difficulties involved in separating embedded content from its container [Fri10].

1.3 Steganalysis

Steganalysis is the practice of detecting the presence of messages that have been hidden using steganography. Ideally, steganalysis will also determine the contents of the message. In the Prisoner's Problem, steganalysis is the job of the Warden. Wardens can be associated with a variety of categories such as: *active*, *passive* and *malicious*.

1.3.1 Passive Warden

A passive warden inspects data, attempting to determine by observation alone, whether or not a message is present. Passive wardens will often use statistical analysis in an effort to ascertain the presence of a message [Fri10].

1.3.2 Active Warden

An active warden will not only attempt to determine the presence of a message, but also prevent the exchange of covert messages.

In the case of Linguistic steganography, an active warden will rephrase passages and exploit synonyms in intercepted communiques. During World War II the U.S Post Office censored the contents of telegrams to ensure that hidden messages were not exchanged. In one instance, a censor changed "father is dead" to "father is deceased", resulting in the reply "is father dead or deceased?" [HLvR+00].

An active warden will only perform slight modification of any intercepted messages.

1.3.3 Malicious Warden

A malicious warden will attempt to catch the prisoners' communicating, often by modifying large portions of the container or even fabricate entire messages by impersonating one of the prisoners [Cra96, BDBG08].

1.4 Structure of this Report

The remainder of this report is divided into eight chapters. In sect. 2 we review the existing literature starting with an overview of the fundamental principles and techniques, before discussing steganography and steganalysis in detail. In sect. 3 we analyse the requirements and goals of this project, and identify how the success of the project will be evaluated. Section 4 presents the progress that has been made thus far. Finally, in sect. 5, we summarise our findings and discuss the avenues for future research. A plan is also proposed providing, in detail, a break down of the project into component sections and the corresponding deadlines for each section.

Chapter 2

Literature Survey

There are numerous techniques for hiding data in a digital container file. Although I will be focusing solely on using a video container file, there are techniques in audio and image steganography that still bear relevance to video file formats. Furthermore, video can be split into two components: the *audio stream* and the *picture stream*. To be able to work with video steganography, it is important that we understand the audio and image (picture) techniques that have already been developed and explored within digital steganography.

2.1 Fundamentals and Background

There are a variety of steganographic techniques that can be used to conceal information in a container file. The steganographic techniques discussed herein fall into one of the following categories which are defined by the method of data hiding: *injection*, *substitution/insertion*, *generation* and *transform domain*.

2.1.1 Injection Techniques

Steganography performed by injection is by far the simplest steganographic technique. As the name suggests, data is injected into redundant areas of the container file. Most files have an EOF (end of file) marker or a file size marker, which indicates where the reading of a file should cease. Data can be placed at the end of the file (after the EOF marker), without affecting the integrity of the container file [Col03]. This technique is very simple and as such, is very easy to detect.

This technique works well on files such as EXEs and WAVs. EXE files have an end of file marker, after which you can place any quantity of data [Col03]. WAV files have a data length defined in their header (see A.1.1), therefore, any data can be injected after the WAV-data section (at the end of the file).

The nature of injection techniques means that it is a fairly straightforward process to detect and extract the covert data. Techniques that embed the data into the container (via generation or insertion) are generally harder to detect because the covert data is interwoven with the original container data, thus making it harder to identify the presence of covert data.

2.1.2 Substitution/Insertion Techniques

A substitution or insertion technique will identify areas of a file of least relevance, and replace this data with the covert data [Col03]. This technique does not modify the size of the container file, and is consequently limited by the steganographic capacity of the file.

LSB Manipulation

One of the most common steganographic techniques is *least significant bit* (LSB) manipulation [JDJ03]. LSB manipulation can be easily applied to some audio and image formats, and works by modifying part of the representation of the data stored within the container format. In the context of audio it is possible to modify the LSB of each sample without causing any audible difference

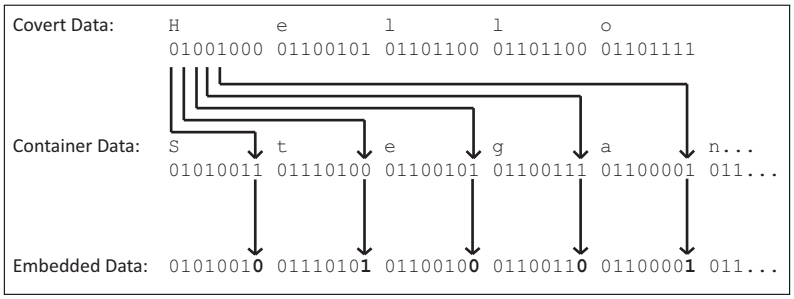

Figure 2.1: *This figure shows LSB encoding of the word "Hello" inside container data.*

to the sound during playback [Col03]. With palette-based image files LSB manipulation works in a similar vein as audio files, but instead of modifying the LSB of a sample, it is the LSB of the three-byte RGB colour representation that is modified. In either case, the fluctuation in colour or sound that is introduced by LSB is not noticeable to a human.

Before explaining how different compression levels affect the use of LSB manipulation the *spatial* and *transform* domains must be defined. The spatial domain in the current context is best defined as the normal image space in which pixels can be represented as a two-dimension matrix. Representing an image in the spatial domain allows for the image to be changed in space by the direct manipulation of pixels. On the other hand, the transform (or frequency) domain exploits the fact that any signal can be represented as a sum of sine waves. In the transform domain an image is represented by the different frequencies that comprise it, and their respective intensities.

With uncompressed and lossless compression formats the exact representation of data is preserved which makes LSB manipulation straightforward [Fri10]. In contrast, lossy compression generally discards insignificant portions of data, such as LSBs, this makes LSB manipulation redundant when work with data that is compressed using lossy compression. Generally compressed data cannot be modified as this corrupts the compressions and has an adverse impact on the data that has been compressed; this rules out applying LSB manipulation to compressed data.

WAV audio files are uncompressed which allows for direct modification of the raw data stored in the files (so long as the file header is preserved). Palette-based images (PNGs and GIFs) are examples of lossless compressed formats. Once the data in these files has been decoded, LSB manipulation can be applied to any of the pixel values. JPEG and MP3 are two examples of lossy compression file formats. JPEG images are represented in the transform domain (see sect. 2.1.4) and the majority of this information is encoded using lossy compression. Small portions of the file (DCT coefficients) are encoded using lossless compression. Generally any data that is lossless encoded can be manipulated using LSB encoding [Fri10].

2.1.3 Generation Techniques [1]

Generation techniques involve generating a container file based on the covert data that is to be embedded. Most generation techniques create fractal images, which have specific mathematical properties; essentially a fractal consists of patterns and lines of different colours.

With a generation technique there is no original container file because the cover object is completely generated, this provides a unique advantage. On the other hand with other steganographic techniques if the original container file exists (or is leaked) outside the secure domain this can provide an attacker with significant information that can accelerate a steganalysis attack.

A steganography system that uses generation techniques should produce a fractal image that fits the profile of those communicating, for instance, if Alice and Bob are car enthusiasts, using pictures of cars has a natural plausibility and will not arouse suspicion. This technique is disadvantages by the fact that producing a steganography system that generates realistic fractals is complex

[1] Much of the information in this section is based on [Col03]

and time consuming. These disadvantages would prevent a generation technique being used in time–critical situations such as real time video steganography.

Research into generation techniques is very limited – this could relate to the fact that the method of encoding data is often heavily dependent on the subject matter of the fractal.

2.1.4 Transform Domain Techniques

Transform domain techniques are generally used on compressed container files. For instance, data hiding in JPEGs is commonly achieved by operating in the frequency domain and modifying the *Discrete Cosine Transform* (DCT) [And96, ZK95, RDB96]. One of the earliest methods for hiding data in JPEG files relied on changing the LSB of the DCT coefficients. This technique is relatively basic, and numerous steganalysis methods have been developed which are easily capable of detecting covert data that is embedded using this method (see 2.3). Other DCT-based methods, such as *F5* [Wes01], *Outguess* [Pro01], *Model-Based* [Sal05, Sal03], *Modified Matrix Encoding* [KDR06] and *Perturbed Quantization* [FGS05] have been developed, all of which are dependent on modifying the DC coefficients.

Discrete Cosine Transform

The discrete cosine transform can be used to convert an image from the spatial domain into the frequency domain. The spatial domain represents data based on intensity of pixels. A steganographic technique that uses LSB manipulation on a palette-based image (PNG of GIF) would be working in the spatial domain, as changing the LSB modifies the pixel colour (intensity). In contrast, DCT separates parts of an image based on frequency. Image signal energy is generally stored in low-frequency regions, therefore high-frequency information can be discarded or manipulated without causing significant degradation of image quality [WJN10]. Steganographic approaches that operate in the transform domain generally use properties of the DCT; there is very limited research in alternative transforms such as Discrete Wavelet Transforms [KK10].

The 2-dimensional DCT, $F(m,n)$, of an $N \times M$ pixel image is defined as follows:

$$F(m,n) = \frac{2}{\sqrt{MN}} C(m)C(n) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\frac{(2x+1)m\pi}{2M} \cos\frac{(2y+1)n\pi}{2N} \tag{2.1}$$

where

$$C(m) = C(n) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } n = 0 \\ 1 & \text{if } n \neq 0 \end{cases} \tag{2.2}$$

As mentioned previosly, LSB manipulation cannot be applied to the colours of pixels when working with lossy compression formats such as JPEGs. This is because JPEG images use a DCT as part of the compression process, during which values such as LSBs are not necessarily retained. Whilst the conversion between the spatial and the transform domain (and vice versa) uses lossy compression, the discrete cosine coefficients are stored using lossless encoding, therefore most JPEG steganography techniques encode data in the discrete cosine coefficients.

2.2 Video Steganography

Video Steganography is now a growing area of research as a video container file has numerous advantages not exhibited by other container formats. Modification of a video file is significantly more difficult to detect by the human visual system, as frames are displayed on screen for extremely brief periods of time [AFJK+10]. Furthermore, video frames are not crisp, sharply focused images, so variations in pixel colour induced by steganography will blend into the frame. Video (especially HD Video) container files are significantly larger than audio or images files, thus reducing the

problem of steganographic capacity. Noda et al. overcome steganographic capacity issues by using a *Bit Plane Complexity Segmentation* (BPCS) technique for wavelet compressed video data [NFNK04]. Similarly, Jalab et al proposes a frame selection method for hiding data in MPEG video, again using BPCS [JZZ09]. BPCS achieves a high embedding rate, with minimal levels of distortion. This is achieved by identifying noisy regions of an image frame, and embedding a high density of covert data. Embedding high proportions of data in already noisy sections of a frame does not cause any significant degradation of image quality [JZZ09, NFNK04].

Eltahir et al. propose and discuss the application of LSB manipulation in the context of video [EKZZ09]. Unfortunately, their paper does not discuss the security of the technique. LSB manipulation, in comparison to other techniques, is relatively easy to detect and more often than not, steganalysis can be quickly performed on the distribution of LSBs to determine the presence of a message (see 2.3. LSB embedding can be made more difficult to detect by encrypting the covert data before embedding it in the container file. The encryption process produces encrypted data with a more uniform distribution across the ASCII range, thus suppressing any strong characteristics of the original covert data (see figure 4.3 and section 4.1.3) [Col03].

Singh et al. have published a video steganography technique that specifically addresses hiding an image in a video [SA10]. They discuss the nature of a video file as a container, stating that rows of pixels that form an image can be spread across the frames that comprise the video. Whilst the proposed method centres around LSB manipulation, this is one of the few papers that exploits the multi-dimensional aspect of a video container file. This paper highlights that without the entire video an attacker will not be able to determine the full meaning of a covert message (but noe, this is not the goal of steganography; the goal is to avoid detection). Singh et al. also claim that the proposed technique is "very useful in sending sensitive information securely" without providing any supporting evidence or justification for this claim, or the effectiveness of the proposed technique.

2.2.1 Transform-Domain

Westfeld and Wolf describe a system that uses a DCT method to embed data in the H.261 standard [WW98]. This method exploits the characteristics of H.261, and similarly, M-JPEG and MPEG. These standards essentially use JPEG images to construct the picture stream. As with JPEG, H.261, MPEG and M-JPEG use a discrete cosine transform as a basis for the lossy compression that they use. Westfeld and Wolf describe a technique for modifying "suitable" DCT blocks. This vetting quality ensures that the encoded message cannot be detected just by analyzing the DCT coefficients: a direct comparison has to be made to the original container file [WW98].

Chae et al. also propose a DCT based method, in which the amount of data to be embedded in each 8×8 DC block is determined by a scale factor. This technique adjusts the scale factor so that more data is hidden in textured areas, an approach based on the understanding that "the human visual system is more sensitive to the changes in low frequency regions than in highly textured regions" [CM99].

2.2.2 Motion Vector

Before proceeding with an explanation of motion vector based techniques, several terms specific to video encoding need to be defined [Muk11]:

Video Terminology

- **Macroblock**
  Macroblocks in video are similar to macroblocks in image encoding. A macroblock is a 16×16 pixels segment in a frame. This forms the basic coding unit.

- **Intra-frames (I-frame)**
  An intra-frame, or I-frame, is coded independently from any other frames. These frames allow a user to seek to a random point in a compressed video stream.

- **Predicted-frames (P-frame)**
  A predicted frame, or P-frame, is calculated by a prediction from its nearest I- or P- frame. The prediction process uses motion compensation, which produces a high level of compression from these frames.

- **Bidirectional-frames (B-frame)**
  A bidirectional frame, or B-frame, is calculated using prediction from frames in both directions (a previous frame and a future frame). Predicting these frames required a higher level of computation, but the resultant frame has a greater level of compression than P-frames.

- **Motion Vector**
  P- and B- frames are based on motion vector values which are coded with respect to a reference frame (a previous P- or B- frame). A frame is broken down into segments called macroblocks. Motion vectors are used to describe the offset position of each macroblock in the current frame, from the position in the reference frame. In the case of a B-frame, motion vectors can describe offsets from a previous frame, a future frame, or both.

Motion Vector Techniques

A large proportion of image and video steganographic techniques involve concealing data in DCT blocks, by varying degrees. Alternatively, Prabhakaran and Shanthi propose a hybrid cryptography-steganography method for hiding an AES encrypted message inside the motion vector of a video [PS12]. This technique is certainly worth noting, as the overall quality of the video file is preserved, and an additional layer of security is provided with use of AES cryptography.

Shanableh builds on the principle of motion vector encoding, by using a layered approach to encode data in the motion vector and quantization scales [Sha12]. This proposed method doubles the steganographic capacity of the container file when compared to other motion vector based methods. This technique can only be used on raw video, however; in the case of compressed video, simply adding a transcoding step will allow for the encoding in both motion vector and quantization scales. The number of quantization scales available in a coded video frame is limited; Shanableh increases this number for each frame by using "multilayer encoding" - two layers, a low-resolution base layer and a higher-resolution enhanced layer, thus providing two quantization scales for each macro block [Sha12]. This technique provides a high steganographic capacity, whilst causing minimal degradation of video quality. This method seems to specifically focus on increasing the steganographic capacity of a video container file, something which is not necessarily esssential, given how one minute of video can contain in excess of a thousand frames.

Fang and Chang document a method that focuses on the embedding of data inside the motion vectors. In their scheme, they propose embedding the data in the phase angle of the motion vector of macroblocks, a technique that works for both compressed and uncompressed video [FC06]. As part of Fang and Chang's proposed method, they select candidate macroblocks for encoding based upon a magnitude threshold of the motion vector, as modifications made to a motion vector of high magnitude (a fast moving object) are relatively undetectable, whereas modifications to small motion vectors are likely to produce noticeable changes.

Aly proposes a different approach to that of Fang and Chang. In Aly's approach he selects candidate macroblocks based on their prediction error, and the covert data is embedded in the LSB of the suitable vectors [Aly11]. Under evaluation this technique has proven successful, and keeps distortion and overhead to a minimum. Both Aly and Fang et al. produce methods that are successful in encoding data with minimal distortion, but their methods have one key difference. Fang and Chang select motion vectors based on properties exhibited by the motion vectors themselves, whereas Aly selects motion vectors based on the properties of the associated macroblocks. Although different, these approaches yield similar results in terms of image quality [Aly11, FC06]. Aly compares his approach with that of Zhang et al. [XPZ06]: his findings show a better balance of the payload (covert data) between P- and B- frames with his method [Aly11]. If time permits, it would be worth investigating how Aly's method responds to different payload sizes.

Motion vector approaches generally embed data based on the properties exhibited by the motion vectors, or their associated marcoblocks. Most of the research into motion vector based approaches deal just motion vectors and their properties. However, as outlined above, Aly's approach of selecting candidate vectors based on their macroblocks is just as successful, and offers slightly better preservation of quality.

2.2.3 Streaming and Real Time

Streaming videos across the internet has become an incredibly popular activity over the last 7 years. The FLV video format was specifically developed for delivering videos across the internet [2]. FLV is significantly simpler than other formats, and Mozo et al. prove that injection-based steganography can be used with the FLV file format [MOR+09] (all documented video steganographic techniques are insertion-based, not injection based). FLV files can contain audio, video and meta blocks, each indicated by a tag, and the technique involves injecting data at the end of a video block. This research has proven that FLV files are highly resilient, and can undergo significant modification without affecting the quality of the video playback. A significant disadvantage with injection-based steganography is the fact that the file size inflates. However, the FLV format is sufficiently resilient that video tags can be removed, and the integrity of the video is maintained. As Mozo et al. prove, it is possible to compensate for the addition of covert data by removing a corresponding number of video tags, although removing too many video tags has been shown to cause some degradation to playback quality.

In comparison Liu et al. proposed a real-time steganographic approach that works with the more complex MPEG-2 file format. Whilst their technique does work, by their own admission it is a fragile technique [LLLL06] that neglects the resilience aspect of steganography. The strengths, weaknesses and approaches of the methods proposed by Mozo et al. and Liu et al. vary significantly. Injection-based methods (such as Mozo et al.) area easier to detect than substitution techniques (Liu et al.) purely based on the fact that injection-based methods modify the file-size – a factor easily noticed without extensive analysis. Given that both methods were specifically suggested for the purpose of streaming video, we would argue that, Liu et al.'s method would be preferable as container file size is not increased by the covert data. Nonetheless, any form of data transmission, including internet streaming is not guaranteed to be free from error; with a fragile steganographic scheme, the slightest error in transmission could significantly impact the success of the communication, and with this consideration Liu et al. approach is disadvantaged.

2.3 Steganalysis

Steganalysis is the art and science of detecting messages that have been hidden in container objects via the application of steganography. In the prisoners' problem (section 1.2.1) it is the Warden's job to use steganalysis to attempt to determine whether a message is present in communication channels. Just like with steganography, there are numerous different methods for steganalysis, but from a theoretical stand-point, for steganalysis to be successful, the results only need to show a higher probability of detection than random guessing [Fri10]. Steganalysis has also been described as "the process of detecting with high probability and low complexity the presence of covert communication through innocuous multimedia distribution" [BK04]. It is perhaps worth noting that steganalysis only needs to detect the presence of a hidden message, the exact process of determining what the embedded message reads is reserved for the field of *forensic steganalysis* [Fri10].

[2] Other formats such as RealMedia were also developed for internet streaming. FLV has become an accepted standard for internet streaming, and websites such as YouTube, Hulu, VEVO, Yahoo! Video, metacafe and Reuters use FLV

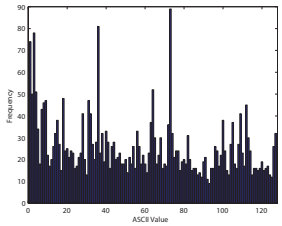

Figure 2.2: *ASCII distribution from LSB string before data is embedded in a PNG image.*

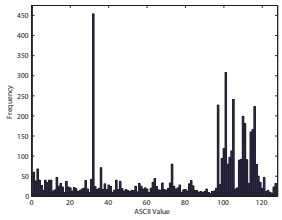

Figure 2.3: *ASCII distribution from LSB string after data is embedded in a PNG image.*

2.3.1 Overview of Steganalysis Techniques

Steganalysis techniques can be split into two different categories:

- Targeted Steganalysis
- Blind Steganalysis

Both targeted and blind steganalysis can use a statistical approach known as *statistical steganalysis*.

Targeted Steganalysis

Targeted steganalysis can be performed when the Warden knows the algorithm or any aspect of the stegosystem that is being used. Targeted steganalysis is generally simpler than blind steganalysis, because the attacker knows some aspects of the mechanics behind the stegosystem.

> **Example**
> Let us assume that we have a stegosystem that encodes a message in the LSB of a palette-based image, in which each pixel is represented by three bytes (one byte for each RGB component).
> A simple targeted attack can be produced to detect this steganographic scheme. By forming a string of the LSB characters, we can then compare the frequency of ASCII characters represented in this string. If a message is encoded we will see a spike corresponding to the frequency of characters that compose the latin alphabet [Col03]. Figures 2.2 and 2.3 illustrate a container before and after the application of the suggested steganographic scheme.
> Compare figures 2.2 and 2.3, notice that in figure 2.3 there is a significant increase in frequency for ASCII values 32, 65-90 and 97-122. These values reflect the *space* character; and upper and lower case A-Z characters. This illustrates that with the simplest form of LSB encoding it is possible to detect the presence of embedded data based on ASCII frequency. This technique is akin to the analysis that is used for substitution ciphers in cryptography. The language of the plain-text (or in this case, covert data) will have language specific properties reflecting, for example, the fact that the letter E is the most common letter in the English language [Sch96].

Blind Steganalysis

In the context of blind steganalysis the Warden does not know anything about the steganographic system that is potentially being used to hide messages in a container object. Blind steganalysis is significicantly more complex as an ideal steganalysis algorithm should be able to detect every possible steganographic scheme [Fri10].

Statistical Steganalysis

Detecting the presence of steganography can be a complex issue, especially when a multi-dimensional container file (such as an image or video) is being used. In reality, the "detection problem" that

is encompassed by steganalysis is resolved by representing the container file as a set of numerical functions [Fri10]. Statistical steganalysis can be used for both blind and targeted steganalysis.

2.3.2 Video Steganalysis

With any stegosystem covert data is embedded in a container file, regardless of whether this is happening in the spatial or transform domain. Zhang et al. identify that covert data is usually of a higher-frequency signal in comparison to the rest of the image; with this knowledge they propose a system for video steganalysis based on aliasing detection. Their method uses *Haar wavelet filters* to distinguish between the container and covert data. With a Haar wavelet filter, a lowpass filter provides an approximation of the container, whereas a highpass is able to extract the higher frequency covert data. Further statistical analysis is conducted using the *Laplacian distribution* [KP03] to distinguish hidden data from the natural container frame. This approach has been shown to yield good results with a low false-negative and false-positive for the tests conducted [ZSZ08].

Video steganalysis is a largely unexplored area, with most research centring around steganographic methods as opposed to steganalysis [CZF12]. Current steganalysis research models videos as images, whereby the embedding process modifies the Gaussian noise of a frame [BKZ06, ZSZ08, PDB09, JKH07]. Steganographic approaches that utilize motion vectors for data hiding are becoming more and more ubiquitous; therefore this model is likely to become less relevant as the properties of motion vectors are being exploited.

Cao et al. propose a method for detecting motion vector based encoding for sub-optimal methods of data hiding. Modification to motion vectors can significantly modify the internal dynamics of video compression. The fundamental principle behind their research relies on decompressing the video to the spatial domain before re-compressing. They argue that this decompression and re-compression cycle is likely to revert the motion vector values back to their unmodified state, where prediction errors can be used to perform statistical analysis at various stages of the re-compression process [CZF12].

2.4 Summary

There is a growing range of techniques for embedding data in video container files, many of them built on the principles of image steganography. For instance, a large number of techniques use DCT coefficients, as such, these techniques are limited to the MPEG video codec. Recently, more versatile techniques such as motion vector based approaches have started to emerge and are quickly becoming commonplace.

Streaming video across the internet has become incredibly popular over the last seven years. Some of the steganographic approaches that have been discussed were proposed specifically for "real time video" and "streaming", however these approaches neglect some aspects which are contextually important.

Steganalysis is mainly a statistical approach, whereby a container file is represented as a set of numeric functions. These functions usually model a container file in part, and will attempt to determine whether a message is embedded within it. The goal of determining the content of a message is reserved for the field of "forensic steganalysis".

Chapter 3

Requirements and Analysis

The main aim of this project is to develop a video steganography system that fully utilises the features of a video container file. The proposed system should be secure and practical. Once a steganographic scheme has been produced, steganalysis will be used to evaluate the performance of the system.

3.1 Project Overview

Video files contain a mass of data, and are naturally large in size. This vast quantity of data, combined with the popularity of sharing video files makes video one of the more attractive digital media containers (as opposed to images and audio files). Large file sizes allow for a large steganographic capacity, and furthermore sharing video files has become so popular that the nature of exchange video files is not a suspicious or unusual activity that would warrant unnecessary attention.

3.1.1 Container File

A video file comprises two core components: the video stream (a sequence of still image frames) and the audio stream. Most steganography systems only focus on embedding data in the video stream and ignore the audio stream completely. With our proposed system both the audio and video components of the file would be utilised.

The audio stream of the container file can be utilised in two possible ways: it could be used to store information, or audio stream data could be used with a steganographic key to determine how the covert data should be encoded. Ideally, the audio stream will contain embedded data. There is limited research in video steganography utilising the audio stream, therefore there may be unforeseen complications with encoding data into the audio stream of a video file. Should this prove untenable we will investigate utilising the audio stream data in conjunction with the steganographic key to determine the exact encoding of the covert data.

Whilst conducting the literature survey I discovered that a large proportion of the steganographic techniques in the literature treat a video frame as a still image, and as such, use DCT methods similar to those used for JPEG images. Whilst this technique works well for MPEG-based formats, our proposed system will ideally work with newer video formats such as H.264 which supports HD video. Depending on the encoding scheme used the higher resolution (larger file size) format of HD could allow for a greater steganographic capacity over other lower-quality resolutions.

3.1.2 Security[1]

There are a couple of security principles that are worth considering: the notion of security being a "trade-off", and Kerckhoff's principle. Schneier reminds us that security is a "trade-off", the more secure a solution the greater the inconvenience it can cause to society. The practical and social implications of a security system should, he argues, always be a key consideration – producing a system that has real-world practical application is certainly a desirable requirement.

[1] For more information, the reader is referred to Schneier's survey [Sch96], on which this section is partly based.

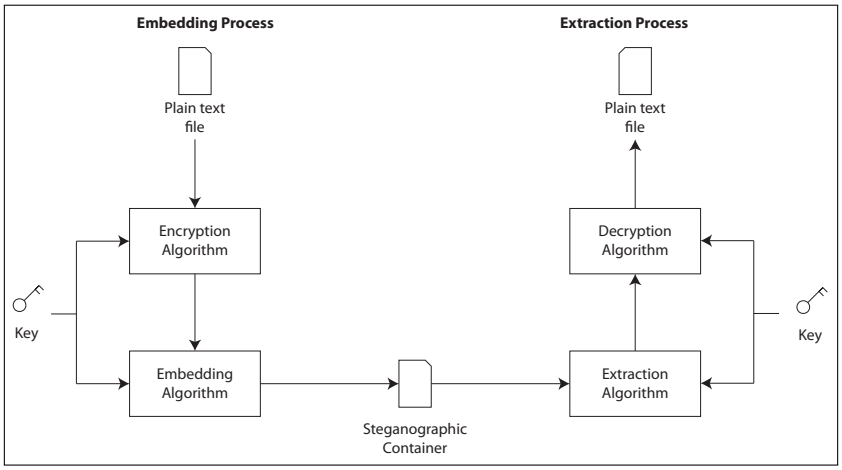

Figure 3.1: *System overview.*

Kerckhoff's principle states that a cryptosystem should remain secure even if everything about the system (excluding keys) is public knowledge. Whilst this principle is inherent to cryptography it can be applied to anything in the domain of security. As such, this project should adopt Kerchoff's principle and produce a stegosystem that is not reliant on obscurity.

Producing a secure solution to any problem is difficult, and the only true test as to how secure it is, is time – Cole argues that if flaws and attacks have not been discovered after a few years, it's probably secure [Col03]. This argument works well for algorithms and systems that are public knowledge, but if an organisation's security system is broken or flawed it is not in their interest to publicise the fact – this makes it difficult to ascertain the security of proprietary and closed-source software solutions.

Security of the container file can be increased by combining steganographic techniques with cryptography so that an encrypted version of the covert data is embedded. Whilst this does not increase the security of the steganographic technique per se, this does add a layer of security to the system without detracting from the usability experience of the end user.

3.2 Steganography System

The proposed steganography system (*stegosystem*) will take a *plain textfile*, *container file* and a *password*, and then superficially the system will encrypt and embed *plaintext* file in the *container file*. Figure 3.1 gives an overview of how the system is expected to embed and extract covert data.

3.2.1 Encryption and Decryption

Encryption will be performed by an implementation of the *Advanced Encryption Standard* (AES) algorithm. The use of AES encryption will add a high level of security to the data being embedded. AES is used by governments throughout the world [DKR05, CMR06].

3.2.2 Embedding Algorithm

During the embedding process, data needs to be encoded in the frames of the video and the audio stream. Since it is possible for a video frame to span multiple packets of a video file, the proposed system will need to read sequential packets of data from the video file to avoid complications that could arise from seeking to random portions of the video file. As explained in section 2.2.2, data will be embedded in P- and B- frames by encoding the information in the motion vectors. LSB

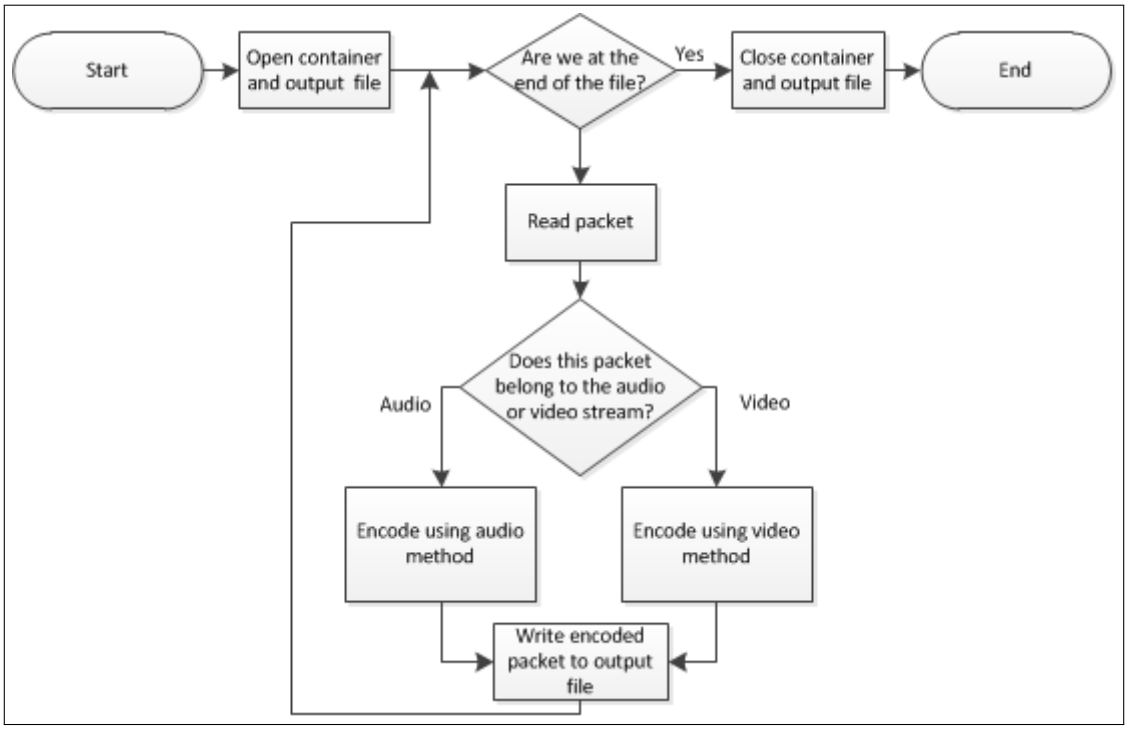

Figure 3.2: *Flowchart illustrating embedding algorithm.*

manipulation, or a similar technique will be used to embed data in the audio packets. The flowchart in figure 3.2 shows how the embedding algorithm should be applied to the container input stream. Further research and investigation into the data structure of an audio stream will be required to identify an exact encoding method.

3.2.3 Extraction Algorithm

The extraction algorithm will work in a similar vein. Packets will be read sequentially from the container file, and the correct extraction method will be used depending on whether the packet contains a video frame or audio data.

3.3 Evaluation

Evaluation of this project will be performed by using steganalysis. A good implementation of a steganographic system should be one whose use is difficult to detect.

Evaluating the effectiveness of the proposed solution will not be easy. We will only consider targeted steganalysis for the purpose of evaluating our proposed steganographic method. Targeted steganalysis will perform a more rigorous evaluation of the system over blind steganalysis given that a targetted approach required some knowledge of the stegosystem, Furthermore, using targeted attacks which exploit knowledge of the stegosystem will demonstrate whether Kerckhoffs's principle has been considered.

It is worth noting that performing steganalysis on our own proposed system is not conclusive. It is certainly feasible that mistakes could be made in the stegosystem, and mirrored mistakes could be made during the steganalysis. In an attempt to reduce the likelihood of this, our proposed solution will be evaluated against other researched steganalysis approaches where possible.

3.3.1 Steganalysis

In traditional steganalysis fashion the suspected container file should be represented by a simple model, and by performing statistical analysis we should be able to determine if covert data is present or not. If a steganalysis method is developed that is able to determine the presence of covert data more accurately than random guessing then the steganographic technique is not secure.

3.3.2 Iteration

Throughout this project an iterative feedback process will be adopted. The intention is to release a prototype stegosystem that implements the simplest steganography scheme before iterating to the next version of the system. Steganalysis will be used to determine the effectiness of the current version; the feedback will then be used to improve the system in the next iteration.

3.4 Functional Requirements

This section identifies the functional requirements of the system that will be necessary in order to achieve the goal of this project. The set of requirements are organised into two groups: *steganography requirements* and *steganalysis requirements*. In each instance requirements are categories as: essential, desirable and optional extras.

3.4.1 Steganography Requirements

Table 3.1 identifies the functional requirements for the steganography aspects of the system.

Requirement	Priority	Description
1.1	Essential	Read meta information from the video file (bitrate, number of frames, etc.).
1.2	Essential	Iterate and parse video frames.
1.3	Essential	Parse audio stream data.
1.4	Essential	Implement embedding algorithms that embed data in the video stream.
1.5	Essential	Use AES cryptography to encrypt the covert data before embedding, and decrypt the covert data after extraction.
1.6	Essential	The embedding algorithms should produce an output video that has no visible or audible different to the original video file.
1.7	Essential	Use a steganographic key to determine the placement of covert data in the container file.
1.8	Desirable	Embedding algorithms should utilise the audio stream. Preferably for the purpose of data embedding, however, if this is not possible the audio stream should be used in conjunction with the steganographic key to determine the exact encoding of the covert data.
1.9	Optional extra	Use user–parameters to influence the embedding (e.g. maximum steganography capacity/easy to detect or low steganographic capacity/difficult to detect).
1.10	Optional extra	Support the embedding of multiple files in a single video container file.
1.11	Optional extra	Compress the covert data to reduce the quantity of data that has to be encoded.

Table 3.1: Functional Requirements – Steganography

3.4.2 Steganalysis Requirements

Table 3.2 identifies the functional requirements for the steganography aspects of the system.

Requirement	Priority	Description
2.1	Essential	Playback of steganographic video file and original video file with the ability to step through frames one at a time and compare the output side-by-side.
2.2	Essential	Compute the difference between images represented by two video frames.
2.3	Essential	Compare and analyse motion vectors.
2.4	Desirable	Generate histograms of pixel colour frequencies.
2.5	Desirable	Audibly compare the sound difference between the original and encoded container file.
2.6	Desirable	Compare data from neighbouring frames to spot uncharacteristic changes.

Table 3.2: Functional Requirements – Steganalysis

3.5 Non-functional Requirements

Table 3.3 identifies the non-functional aspects of the proposed steganography system.

Requirement	Priority	Description
3.1	Essential	Runs on Linux operating systems.
3.2	Essential	The application should be easy to use.
3.3	Essential	The User Interface should be clean, intuitive and unambiguous.
3.4	Essential	Progress of the embedding and extraction process should be clearly indicated to the user.
3.5	Essential	Embedding algorithms should allow for high steganographic capacity.
3.6	Desirable	Runs on Windows operating systems.
3.7	Desirable	The embedding process should be fast.
3.8	Desirable	The user can select their preferred embedding scheme.
3.9	Desirable	Warn the user is the steganographic capacity of a container file is too small before embedding covert data.
3.10	Optional extra	Display video meta data to the user.

Table 3.3: Non–Functional Requirements

3.6 Summary

In this chapter we have discussed the aims and requirements of this project. Both functional and non-functional aspects of the system have been outlined in detailed, as well as discussing the security consideration of our proposed system. Finally, we discuss an iterative development model in which we use steganalysis to continually evaluate the system between iterations.

Chapter 4

Progress

Progress thus far has been slower than intended, but an understanding of the fundamental principles of steganography has been established. During the preliminary research for this project two tools were developed that allowed for data to be embedded in WAV audio files and image files. Experimental systems for encoding and manipulating video have also been developed, however video files have a complex structure which has caused some set backs.

4.1 Preliminary Research

Originally starting in Java, two tools were produced that enable the user to embed a file in an image or audio file.

4.1.1 Audio Tool

The audio tool supports the use of WAV files as containers[1]. WAV files have a simple file structure and do not use compression. This tool was capable of reading meta data from the header of the file, and used this information to encode the plain text file into the LSB of each sample. This approach was successful. Whilst the steganographic capacity of the file was not great, this method allowed data to be encoded without a noticeable change in the quality of the audio.

Our technique was evaluated using Shazam[2] – a music identification service capable of identifying a music track from the playback of the file. A variety of music tracks were selected, these were first tested to verify that the service was capable of correctly identifying the tracks. Our audio tool was then used to embed a draft version of this document (11,000 words) into each audio file. The files were then re-tested against Shazam, and each time Shazam correctly identified the song. The source code for this technique is listed in section A.1.2.

4.1.2 Image Tool

The image too [3] is capable of reading files in any popular image format (PNG, GIF, JPEG, BMP) and using LSB manipulation to embed data in the bytes that represented the RGB colour values for each pixel. This tool will only output a PNG (palette based) image format, as raw LSB manipulation techniques do not work with the type of compression employed by image formats such as JPEG. Figures 4.1 and 4.2 show images before and after data has been embedded. There is no noticable different between figures 4.1 and 4.2. Using steganalysis it is possible to detect the presence of a message – figures 2.2 and 2.3 show the distribution of ASCII values represented in the string of all LSBs for figures 4.1 and 4.2 respectively. The source code for this tool is provided in code listing A.3.

[1] An online version of our audio tool is available at `http://steganosaur.us/tools/audio`. This online tool accepts a maximum file size of 2MB.

[2] `http://www.shazam.com`

[3] An online version of our image tool is available at `http://steganosaur.us/tools/image`. This online tool accepts a maximum file size of 2MB.

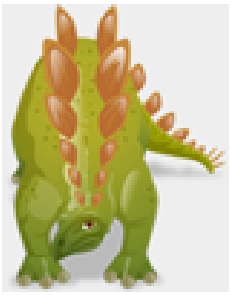

Figure 4.1: Graphic before data is embedded

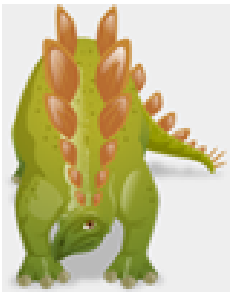

Figure 4.2: Graphic after data is embedded

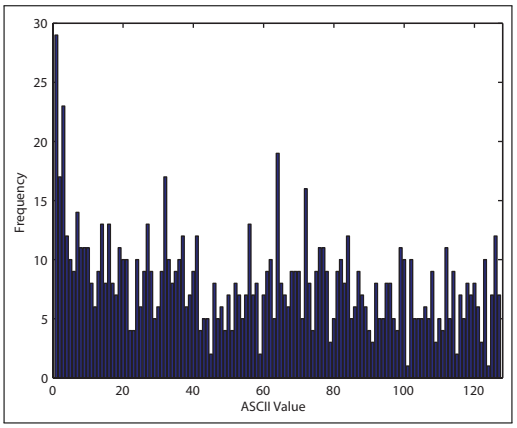

Figure 4.3: *ASCII distribution from LSB string after encrypted data is embedded in a PNG image.*

4.1.3 Steganalysis Tool

In sect. 2.3.1, an example was given of a steganalysis technique that analyses the frequency of ASCII values that are represented in the string of all LSBs of the pixel values in a palette-based image file. Figures 2.2 and 2.3 illustrate how this technique can be used to detect LSB encoding in a palette-based image. Further experimentation has shown that by encrypting the covert data, the presence of embedded data is better disguised, as demonstrated in figure 4.3. The distribution of ASCII values in figure 4.3 is more evenly distributed in comparison to figure 2.3 which shows the unencrypted distribution. Notice that with the unencrypted distribution there is a significant spike in the frequency of ASCII values where the *space* and *A-Z* characters occur.

Figures 2.2, 2.3 and 4.3 are constructed from data that was collected by our steganalysis tool – see section B.1 and figures B.6 and B.7.

4.2 Video Manipulation

As with all of the tools described in the previous section, work on video manipulation was started in *Java* using a library called *Xuggler*[4]. Xuggler is a Java API for video that is based on a native *C* library called *FFmpeg*[5]. After learning how to extract basic meta data from the video file, experimentation progressed to video manipulation. Initially, progress was good; figures 4.4, 4.5 and 4.6 show frames from an inversed and watermarked video in comparison to the original. Whilst this initial work was successful, it was limited in the fact that it only allowed for manipulation of the frame image within the spatial domain. After some extensive research it was concluded that

[4] `http://www.xuggle.com/xuggler`
[5] `http://ffmpeg.org/`

Figure 4.4: Original video frame

Figure 4.5: Visually watermarked video frame

Figure 4.6: Inverted video frame

Xuggler can provide an excellent range of high-level functionality, however this project requires manipulating audio and video data at a much lower-level. Code listing 4.1 shows how Xuggler made it possible to work with frames in the spatial domain, by exposing a `getImage()` method via the `IVideoPIctureEvent` interface.

```
package us.steganosaur;
import com.xuggle.mediatool.*;
import com.xuggle.mediatool.event.IVideoPictureEvent;
import java.awt.*;
import java.awt.image.BufferedImage;
public class VideoPictureInverter {
    public static void invert(final String videoFilename, final String
        outputFilename) {
        final IMediaReader mediaReader = ToolFactory.makeReader(videoFilename);
        mediaReader.setBufferedImageTypeToGenerate(BufferedImage.TYPE_3BYTE_BGR);
        final IMediaWriter mediaWriter = ToolFactory.makeWriter(outputFilename,
            mediaReader);
        final IMediaTool imageMediaTool = new InvertAdapter();
        mediaReader.addListener(imageMediaTool);
        imageMediaTool.addListener(mediaWriter);
        while (mediaReader.readPacket() == null);
    }
    private static class InvertAdapter extends MediaToolAdapter {
        @Override
        public void onVideoPicture(final IVideoPictureEvent event) {
            BufferedImage image = event.getImage();
            Graphics graphics = image.getGraphics();
            for (int i = 0; i < (image.getWidth() * image.getHeight()); i++) {
                int x = i % image.getWidth();
                int y = i / image.getWidth();
                Color originalColor = new Color(image.getRGB(x,y));
                Color newColor = new Color(255 - originalColor.getRed(), 255 -
                    originalColor.getBlue(), 255 - originalColor.getGreen());
                graphics.setColor(newColor);
                graphics.drawLine(x, y, x, y);
            }
            super.onVideoPicture(event);
        }
    }
}
```

Listing 4.1: VideoPictureInverter Java class - inverting a video frame with Xuggler

The inability to continue with Xuggler caused a set back. The range of audio/video manipulation libraries is very limited – most likely due to the complex nature of audio and video files. Most manipulation libaries act as wrappers for FFmpeg – just like Xuggler. Unfortunately, of these libraries, Xuggler was the most advanced, so we decided that the best way forwards from this situation was to switch to using *C* and interacting directly with the native FFmpeg library - without going through a wrapper.

The FFmpeg API has an `AVFrame` data structure[6] that provides access to low level aspects of encoding such as the motion vector table and DCT coefficients. This low-level access is not possible with Xuggler and other FFmpeg wrappers.

With no prior experience of using *C* or *makefiles*, the learning curve was steep. Compiler issues, makefile issues and library linking errors caused significant delay to the project's progress (see B.2 for our solution). The learning curve has been made significantly steeper by the fact that whilst the API documentation on how to use FFmpeg is good, there appears to be limited/no code examples available.

4.2.1 Original Java System

The aforementioned image and audio tools, and the initial video manipulation work were implemented in a single Java application. Screenshots of the GUI can be found in appendix B.1. The GUI that was developed was not comprehensive of all the functionality that was implemented. Despite no GUI for their functionality the following classes were also implemented in addition to the audio and image tools:

- **us.steganosaur.steganography.video.LSB**
  This class housed an unsuccessful LSB manipulation scheme for embedding and extracting data from a video frame. LSB manipulation is not resilient enough to withstand the lossy compression used in video (see section 2). This LSB class was based on the image tool produced during our preliminary research.
- **us.steganosaur.VideoPictureInverter**
  This class inverts all of the colours in a video (see figure 4.6 and code listing 4.1.
- **us.steganosaur.VideoWatermarker**
  This class applies an image watermark to the bottom right of a video (see figure 4.5).

4.2.2 C System

Currently, our focus is on producing a transcode shell[7] that iterates over the packets of the input file. Once this shell is in place work can commence on the new embedding and extraction algorithms.

4.3 Summary

Preliminary research on fundamental steganographic techniques has been undertaken, and successful implementations of some of these techniques have been implemented in *Java* and/or *C*. Experimentation with video manipulation was originally conducted with *Java* and *Xuggler*, however limitations of *Xuggler* meant that this approach was abandoned. Currently, the project focus is on creating a transcoding skeleton in *C* which can be built upon once an underlying framework has been developed.

[6] `http://ffmpeg.org/doxygen/trunk/structAVFrame.html`

[7] The transcode shell will read a video input file packet by packet and will output the exact same video to a different file using the FFmpeg API to manage the video and audio streams.

Chapter 5

Conclusion and Project Plan

The current progress achieved so far indicates that working with a complex file format such as video does have its challenges. Having conducted extensive background reading and experimentation with fundamental steganographic techniques the next stage in the project is to produce the initial prototype of the steganographic system. As outlined in detail in section 3, the final system should comprise the following components:

- **AES Cryptosystem** An implementation of the AES will be used to encrypt the covert data before it is encrypted. The embedded data will also need to be decrypted upon extraction.
- **Transcoding Mechanism** The transcoding mechanism will comprise the main shell of the system. Typically, transcoding is the conversion of one encoding to another. In this context we will be reading frames and packets, and modifying the contents before re-encoding them into the same encoding format.
- **Embedding Algorithm** The embedding algorithm will determine how the covert data is embedded in the container file. Specific embedding methods will be required for each type of data that is being parsed:
  - Audio packet method
  - Video-frame method
- **Extraction Algorithm** The extraction algorithm will retreive the embedded data from the container file. Specific extraction methods will be required for each type of data that is being parsed:
  - Audio packet method
  - Video frame method

Initially, a simple prototype system will be developed that implements only core functionality:

- **Transcoding Mechanism**
- **Embedding Algorithm**
  - Video-frame method
- **Extraction Algorithm**
  - Video-frame method

Once the prototype has been produced an iterative development-steganalysis process will be started. Each successive iteration will look to improve on flaws highlight by the previous steganalysis attempts. AES and audio stream functionality will also be added during later iterations.

Basic work has been started on the trancoding mechanism. Currently, the system is able to take an MP4 container file and extract the video stream into a raw video file. Despite the fact that the transcoding process should convert from a video container file to another video container file, the process of converting from a video container file to raw video data is nevertheless a reasonable start.

5.1 Project Plan

The Gantt chart in figure 5.1 and 5.2 provides an overview of the project roadmap going forwards. To summarise:

- **Prototyping**: 31 December - 23 January
  Develop a prototype system that includes a basic embedding an extraction algorithm
- **Iteration Phase**: 24 January - 13 March
  Each iteration phase will last 6 days, with time split equally between system development and enhancement.
- **Testing**: 14 March - 15 March
  Where possible, testing should be performed as the system is being developed. Where possible the *Check*[1] unit testing framework should be used. Over these two days in-depth testing of the entire system should also take place.
- **Evaluate Project**: 18 March - 22 March
  Evaluate the success of the project in preparation for the final report.
- **Write Report**: 14 March - 1 May
  Write the final report for the deadline on 1 May.
- **Produce Poster**: 1 May - 8 May
  Prepare poster for the poster session on 8 May.

[1] `http://check.sourceforge.net/`

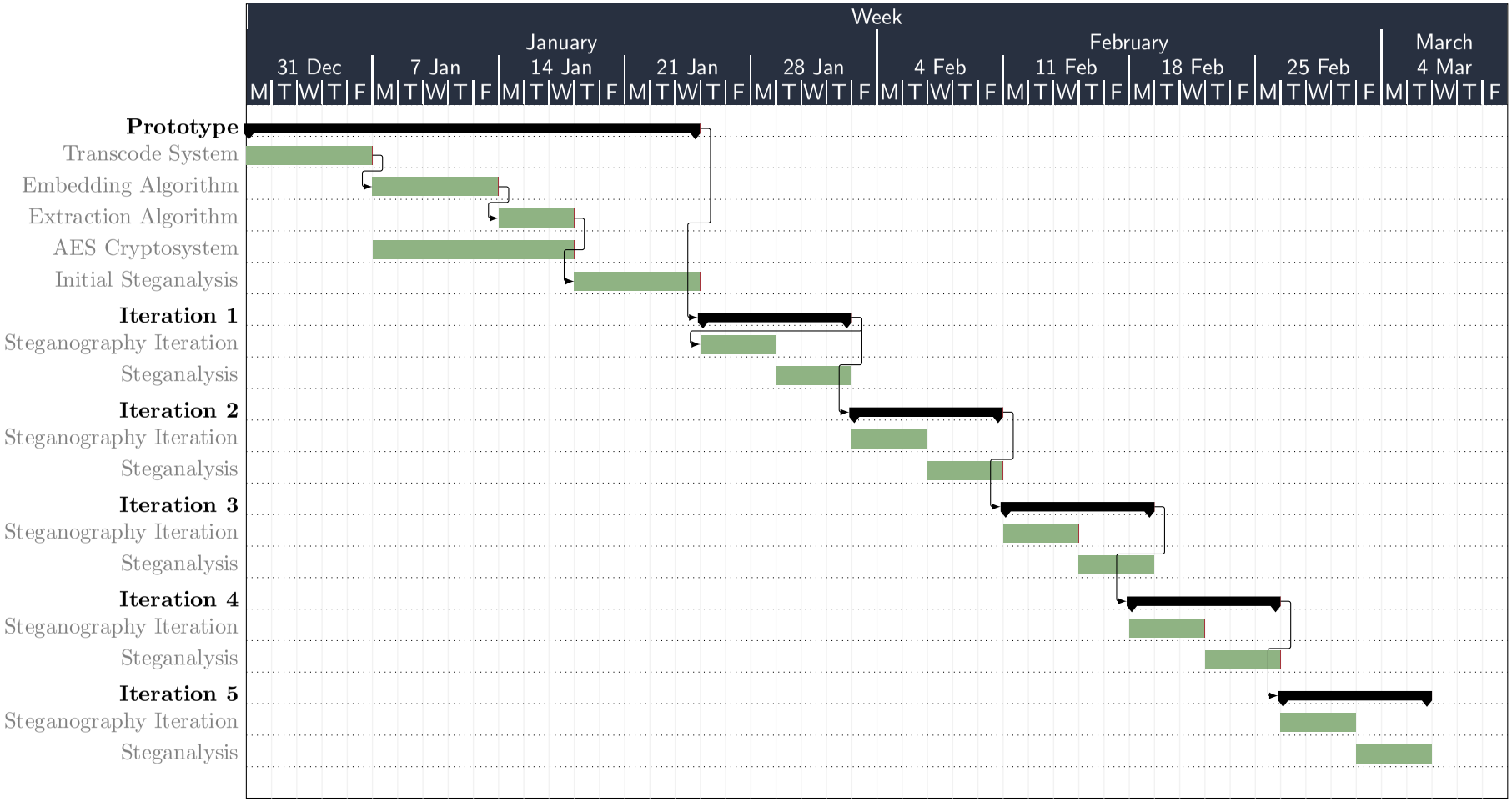

Figure 5.1: Gantt Chart - Part 1 of 2

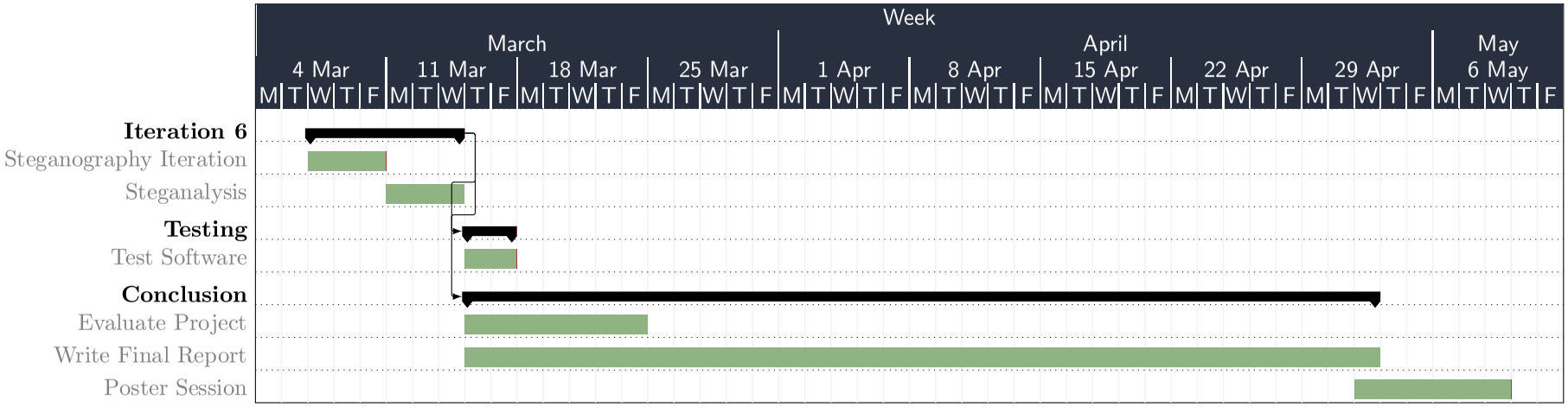

Figure 5.2: Gantt Chart - Part 2 of 2

Bibliography

[AFJK+10] A. K. Al-Frajat, H. A. Jalab, Z. M. Kasirun, A. A. Zaiden, and B. B. Zaiden. Hiding Data in Video File: An Overview. *Journal of Applied Sciences*, 10:1644–1649, 2010.

[Aly11] H. A. Aly. Data Hiding in Motion Vectors of Compressed Video Based on Their Associated Prediction Error. *Information Forensics and Security, IEEE Transactions on*, 6(1):14–18, march 2011.

[And96] R. Anderson. Stretching the Limits of Steganography. *IEEE Journal of Selected Areas in Communications*, 16:474–481, 1996.

[Bac40] F. Bacon. *Of the advancement and proficiencie of learning, or, The partitions of sciences*. Leon Lichfield, Oxford, for R. Young and E. Forest, 1640.

[BDBG08] S. Braci, C. Delpha, R. Boyer, and G. L. Guelvouit. Informed Stego-schemes in Active Warden Context: Tradeoff between Undetectability, Capacity and Resistance, 2008.

[BK04] U. Budhia and D. Kundur. Digital Video Steganalysis Exploiting Collusion Sensitivity. In Edward M. Carapezza, editor, *Proc. SPIE Sensors, Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense*, volume 5403, pages 210–221, Orlando, Florida, 2004.

[BKZ06] U. Budhia, D. Kundur, and T. Zourntos. Digital Video Steganalysis Exploiting Statistical Visibility in the Temporal Domain. *IEEE Transactions on Information Forensics and Security, Vol. 1, No. 4*, 1(4):502–516, 2006.

[CM99] J. J. Chae and Manjunath. Data hiding in Video. In *6th IEEE International Conference on Image Processing (ICIP'99)*, volume 1, pages 311–315, Oct 1999.

[CMR06] C. Cid, S. Murphy, and M. Robshaw. *Algebraic Aspects of Advanced Encryption Standards*. Springer, 2006.

[Col03] E. Cole. *Hiding in Plain Sight: Steganography and the Art of Covert Communication*. Wiley Publishing, Inc., 2003.

[Cra96] S. Craver. On Public-key Steganography in the Presence of an Active Warden. In *Information Hiding, Second International Workshop*, pages 355–368. Springer, 1996.

[CZF12] Y. Cao, X. Zhao, and D. Feng. Video Steganalysis Exploiting Motion Vector Reversion-Based Features. *IEEE Signal Processing Letters*, 19:35–38, 2012.

[DKR05] H. Dobbertin, L. Knudsen, and M. Robshaw. The Cryptanalysis of the AES - A Brief Survey. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *Advanced Encryption Standard - AES*, volume 3373 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2005.

[EKZZ09] M. E. Eltahir, L. M. Kiah, B. B. Zaidan, and A. A. Zaidan. High Rate Video Streaming Steganography. In *Proceedings of the 2009 International Conference on Future Computer and Communication*, ICFCC '09, pages 672–675, Washington, DC, USA, 2009. IEEE Computer Society.

[FC06] D. Fang and L. Chang. Data hiding for digital video with phase of motion vector. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 1422–1425, may 2006.

[FGS05] J. Fridrich, M. Goljan, and D. Soukal. Perturbed quantization steganography. *Multimedia Systems*, 11(2):98–107, 2005.

[Fri10] J. Fridrich. *Steganography in Digital Media: Principles, Algorithms and Applications*. Cambridge University Press, 2010.

[Her96] Herodotus. *The Histories*. Penguin Books, 1996.

[HLvR+00] A. Hanjalic, G. C. Langelaar, P. M. B. van Roosmalen, J. Biemond, and R. L. Lagendijk. *Image and Video Databases: Restoration, Watermarking and Retrieval*. Advances in Image Communication. Elsevier Science, 2000.

[IM04] IBM and Microsoft. Multimedia Programming Interface and Data Specifications 1.0. `http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/Docs/riffmci.pdf`, 2004.

[JDJ03] N. F. Johnson, Z. Duric, and S. Jajodia. *Information Hiding: Steganography and Watermarking - Attacks and Countermeasures (Advances in Information Security)*. Kluwer Academic Publishers, 2003.

[JKH07] J. S. Jainsky, D. Kundur, and R. Halverson. Towards digital video steganalysis using asymptotic memoryless detection. In *Proceedings for the 9th workshop on multimedia & security*, pages 161–168. ACM, 2007.

[JZZ09] H. A. Jalab, A. A. Zaidan, and B. B. Zaidan. Frame Selected Approach for Hiding Data within MPEG Video Using Bit Plane Complexity Segmentation. *Journal of Computing*, 1(1):108–113, 2009.

[Kah67] D. Kahn. *The codebreakers: the story of secret writing*. Macmillan, 1967.

[KDR06] Y. Kim, Z. Duric, and D. Richards. Modified Matrix Encoding Technique for Minimal Distortion Steganography. In *Information Hiding*, volume 4437, pages 314–327. Springer, 2006.

[KK10] V. Kumar and D. Kumar. Performance evaluation of DWT based image steganography. In *IEEE International Advance Computing Conference*, pages 223–238, 2010.

[KP03] T. J. Kozubowski and K. Podgrski. Log-Laplace distributions. *Internat. Math. J*, 3:467–495, 2003.

[LLLL06] B. Liu, F. Liu, B. Lu, and X. Luo. Real-time steganography in compressed video. In *Proceedings of the 2006 international conference on Multimedia Content Representation, Classification and Security*, MRCS'06, pages 43–48, Berlin, Heidelberg, 2006. Springer-Verlag.

[MOR+09] A. J. Mozo, M. E. Obien, C. J. Rigor, D. F. Rayel, K. Chua, and G. Tangonan. Video steganography using Flash Video (FLV). In *Instrumentation and Measurement Technology Conference, 2009. I2MTC '09. IEEE*, pages 822–827, may 2009.

[Muk11] J. Mukhopadhyay. *Image and Video Processing in the Compressed Domain*. CRC Press, 2011.

[NFNK04] H. Noda, T. Furuta, M. Niimi, and E. Kawaguchi. Application of BPCS steganography to wavelet compressed video. In *Image Processing, 2004. ICIP '04. 2004 International Conference on*, volume 4, pages 2147–2150, oct. 2004.

[PDB09] V. Pankajakshan, G. Doerr, and P. K. Bora. Detection of motion-incoherent components in video streams. *IEEE Transactions on Information Forensics and Security*, 4:49–58, 2009.

[Pro01] N. Provos. Defending Against Statistical Steganalysis. In *10th USENIX Security Symposium*, pages 323–335, 2001.

[PS12] B. Prabhakaran and D. Shanthi. A New Cryptic Steganographic Approach using Video Steganography. *International Journal of Computer Applications*, 49(7):19–23, 2012.

[Rab04] K. Rabah. Steganography - The Art of Hiding Data. *Information Technology Journal*, 3(3), 2004.

[RDB96] J. J. K. Ó Ruanaidh, W. J. Dowling, and F. M. Boland. Watermarking digital images for copyright protection. *Vision, Image and Signal Processing, IEE Proceedings*, 143(4):250–256, aug 1996.

[SA10] S. Singh and G. Agarwal. Hiding image to video: A new approach of LSB replacement. *Internataional Journal of Engineering Science and Technology*, 2(12):6999–7003, 2010.

[Sal03] P. Sallee. Model-Based Steganography. In *International Workshop on Digital Watermarking*, volume 2939, pages 154–167. Springer, 2003.

[Sal05] P. Sallee. Model–Based Methods For Steganography And Steganalysis. *International Journal of Image and Graphics*, 5(1):167–189, 2005.

[Sch96] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., 1996.

[Sha12] T. Shanableh. Matrix encoding for data hiding using multilayer video coding and transcoding solutions. *Signal Processing: Image Communication*, 27(9), 2012.

[Sim83] G. J. Simmons. The Prisoners' Problem and the Subliminal Channel. In *CRYPTO*, pages 51–67, 1983.

[Tac90] A. Tacticus. *How to survive under siege*. Oxford: Clarendon, 1990.

[Wes01] A. Westfeld. F5 – a steganographic algorithm: High capacity despite better steganalysis. In *4th International Workshop on Information Hiding*, pages 289–302. Springer-Verlag, 2001.

[WJN10] E. Walia, P. Jain, and N. Navdeep. An Analysis of LSB & DCT based Steganography. *Global Journal of Computer Science and Technology*, 10(1):4–8, 2010.

[WW98] A. Westfeld and G. Wolf. Steganography in a Video Conferencing System. In *Information Hiding*, volume 1525 of *Lecture Notes in Computer Science*, pages 32–47. Springer Berlin Heidelberg, 1998.

[XPZ06] C. Xu, X. Ping, and T. Zhang. Steganography in Compressed Video Stream. In *Innovative Computing, Information and Control, 2006. ICICIC '06. First International Conference on*, volume 1, pages 269–272, 30 2006-sept. 1 2006.

[ZK95] J. Zhao and E. Koch. Embedding Robust Labels into Images for Copyright Protection. In Klaus Brunnstein and Peter Paul Sint, editors, *Intellectual Property Rights and New Technologies, Proceedings of the KnowRight 95 Conference, 21.-25.8.1995, Wien, Austria*, volume 82 of *books@ocg.at*, pages 242–251. Austrian Computer Society, 1995.

[ZSZ08] C. Zhang, Y. Su, and C. Zhang. Video steganalysis based on aliasing detection. *Electronic Letters*, 44(13), 2008.

Appendix A

Fundamental Steganography Techniques

Whilst undertaking preliminary research for this project we experimented with basic Image and Audio steganographic techniques. This section details some of our findings and the tools we produced during this preliminary phase.

A.1 Audio Steganography

As with video, there are numerous audio formats, each with their own specific properties. WAV (Waveform Audio File Format) is an uncompressed audio format. Naturally, uncompressed formats are a lot easier to work with in comparison to compressed formats such as MP3.

During our exploration of Audio Steganography we implemented a Steganography tool capable of encoding data in a WAV file, this tool encoded data in the LSB of each sample. This is the simplest and most discreet substituion-based steganographic technique for audio files.

A.1.1 WAV File Format

The table below outlines the structure of a WAV file. A WAV file is split into a *header* and *data* portion. The first 44 bbytes of a WAV file contain the fixed–length header [IM04].

Position	Description	Example Value
01-04	Indicate that the file is a RIFF file.	"'RIFF"'
05-08	Size of the entire file. (Usually specified once the file has been created)	*integer*
09-12	File type	"'WAVE"'
13-16	Format chunk marker, includes trailing null character.	"'fmt "'
17-20	Size of format chunk	16
21-22	Type of format	1
23-24	Number of channels	2
25-28	Sample rate	44100
29-32	Byte rate = (SampleRate * NoChannels * BitsPerSample) / 8	176400
33-34	Block alignment = (NoChannels * BitsPerSample) / 8	4
35-36	Bits per sample	16
37-40	Data section indicator	"'data"'
41-44	Size of the data section	*integer*

Table A.1: WAV File Format

A.1.2 Source Code

C

Code listing A.1 contains the source code for the Java version of the audio steganography tool. This tool can encode data into the LSB of each sample.
Note: The C version does not support AES cryptography.

```c
#include <stdio.h>
#include <stdlib.h>
#include "wav_audio_steganography.h"
int WAV_HEADER_SIZE = 44;

int file_exists(const char* fileName) {
  FILE *f;
  f = fopen(fileName, "r");
  if (f) {
    return 1;
  }
  return 0;
}


int chars_to_int(const char* chars, int charsLen) {
  int intVal = 0;
  int i;
  for (i = 0; i < charsLen; i++) {
    intVal = (intVal << 8) + chars[i];
  }
  return intVal;
}

char* int_to_chars(int intVal, char* charSize) {
  //char charSize[4];
  charSize[0] = (char) (intVal >> 24);
  charSize[1] = (char) (intVal >> 16);
  charSize[2] = (char) (intVal >> 8);
  charSize[3] = (char) intVal;
  return charSize;
}

char* encode_chars(char* container, int containerSize, const char* toEncode, int
    encodeLength, int bytesPerSample, int offset) {
  int i, j;
  for (i = 0; i < encodeLength; i++) {
    int byteValue = toEncode[i];
    // Loop through each bit of the byte
    for (j = 7; j >= 0; j--, ++offset) {
      // Get the bit value
      int bit = (byteValue >> j) & 1;
      // Set LSB to bit
      int pos = (offset + 1) * bytesPerSample;
      container[pos] = ((container[pos] & 0xFE) | bit);
    }
  }
  return container;
}

void wav_stego_encode(const char* containerFn, const char* plaintextFn, const char*
     outputFn) {

  FILE *containerFile;
  FILE *plaintextFile;
  FILE *outputFile;

  // Open container file
  containerFile = fopen(containerFn, "r");
```

```
58
    // Get container file size and read entire data
60  fseek(containerFile, 0, SEEK_END);
    int containerSize = ftell(containerFile);
62  fseek(containerFile, 0, SEEK_SET);
    char* containerData;
64  containerData = calloc(1, containerSize + 1);
    fread(containerData, containerSize, 1, containerFile);
66
    // Bits per sample characters
68  fseek(containerFile, 32, SEEK_SET);
    char* bpsChars;
70  bpsChars = calloc(1, 3);
    fread(bpsChars, 2, 1, containerFile);
72  int bytesPerSample = chars_to_int(bpsChars, 2);

74  // Read plain text file data and file size
    plaintextFile = fopen(plaintextFn, "r");
76  fseek(plaintextFile, 0, SEEK_END);
    int plaintextSize = ftell(plaintextFile);
78  fseek(plaintextFile, 0, SEEK_SET);
    char* plaintext;
80  plaintext = calloc(1, plaintextSize + 1);
    fread(plaintext, plaintextSize, 1, plaintextFile);
82
    // Encode message length
84  char* messageLength;
    int_to_chars(plaintextSize, messageLength);
86
    containerData = encode_chars(containerData, containerSize, messageLength, 4,
        bytesPerSample, 0);
88  containerData = encode_chars(containerData, containerSize, plaintext,
        plaintextSize, bytesPerSample, 32);

90  // Output modified container data
    outputFile = fopen(outputFn, "w");
92  fwrite(containerData, containerSize, 1, outputFile);
    fclose(outputFile);
94
    fclose(plaintextFile);
96  fclose(containerFile);
}
98
void wav_stego_decode(const char* containerFn, const char* outputFn) {
100
  FILE *containerFile;
102 FILE *outputFile;

104 containerFile = fopen(containerFn, "r");

106 // Get container file size and read entire data
    fseek(containerFile, 0, SEEK_END);
108 int containerSize = ftell(containerFile);
    fseek(containerFile, 0, SEEK_SET);
110 char* containerData;
    containerData = calloc(1, containerSize + 1);
112 fread(containerData, containerSize, 1, containerFile);

114 // Bits per sample characters
    fseek(containerFile, 32, SEEK_SET);
116 char* bpsChars;
    bpsChars = calloc(1, 3);
118 fread(bpsChars, 2, 1, containerFile);
    int bytesPerSample = chars_to_int(bpsChars, 2);
120
    int length = 0;
122 int offset = 32;
```

```
  int i;

  // Determine the length of the embedded message by reading
  // the LSBs of the first 32 bytes.
  for (i = 0; i < 32; i++) {
    int pos = ((i + 1) * bytesPerSample);
    length = (length << 1) | (containerData[pos] & 1);
  }

  // Try and check for a valid length - the user may try and decode
  // a file that does not have a message encoded in it.
  if (length < 1 || (length + 32) > containerSize) {
    printf("Error: invalid length of encoded data (%d)\n", length);
    return;
  }

  // Extract message
  char* message;
  for (i = 0; i < length; ++i) {
    int j;
    for (j = 0; j < 8; ++j, ++offset) {
      int pos = ((offset + 1) * bytesPerSample);
      message[i] =  (char)((message[i] << 1) | (containerData[pos] & 1));
    }
  }

  // Output file
  outputFile = fopen(outputFn, "w");
  fwrite(message, length, 1, outputFile);
  fclose(outputFile);

  fclose(containerFile);
}
```

Listing A.1: C version of the audio steganography tool

Java

Code listing A.2 contains the source code for the Java version of the audio steganography tool. This tool can encode data into the LSB of each sample.

```
package us.steganosaur.steganography;

import us.steganosaur.cryptography.AES;
import us.steganosaur.utils.FileUtils;
import us.steganosaur.utils.Utils;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;

public class AudioSteg {

    private static final int WAV_HEADER_SIZE = 44;

    private static byte[] encodeBytes (byte[] container, byte[] toEncode, int
        bytesPerSample, int offset) {
        // Check that the message is not too long for the container
        if (((toEncode.length + offset) * bytesPerSample) > ((container.length -
            WAV_HEADER_SIZE) / bytesPerSample)) {
            new Throwable(new Exception("Message too long for container"));
        }
        for (int i = 0; i < toEncode.length; i++) {
            int byteValue = toEncode[i];
            // Loop through each bit of the byte
            for (int j = 7; j >= 0; j--,++offset) {
```

```
                // Get the bit value
                int bit = (byteValue >> j) & 1;
                // Set LSB to bit
                int pos = ((offset + 1) * bytesPerSample) ;//- 1;
                container[pos] = (byte) ((container[pos] & 0xFE) | bit);
            }
        }
        return container;
    }

    public static void encode(String inputFile, String containerFile, String
        outputFile, String password) {

        // Load input message, encrypt and convert to bytes.
        byte[] message = null;
        try {
            message = FileUtils.getBytes(inputFile);
            if (!password.isEmpty()) {
                message = AES.encrypt(message,password);
            }
        } catch (Exception ex) {
            System.out.println("Ex1: " + ex.getMessage());
        }

        FileInputStream containerStream = null;
        FileOutputStream outputStream = null;
        try {
            // Create file streams
            containerStream = new FileInputStream(containerFile);
            outputStream = new FileOutputStream(outputFile);

            // Copy the header to the output file
            byte[] wavHeader = new byte[32];
            containerStream.read(wavHeader);
            outputStream.write(wavHeader);

            // Get bits per sample
            byte[] bitsPerSample = new byte[2];
            containerStream.read(bitsPerSample);
            outputStream.write(bitsPerSample);

            wavHeader = new byte[10];
            containerStream.read(wavHeader);
            outputStream.write(wavHeader);

            // Read remainder of container
            int bytesToRead = (int) new File(containerFile).length() -
                WAV_HEADER_SIZE;
            byte[] containerBytes = new byte[bytesToRead];
            containerStream.read(containerBytes, 0, containerBytes.length);

            int numBitsPerSample = Utils.bytesToInteger(bitsPerSample);
            int numBytesPerSample = numBitsPerSample / 8;

            // Encode size of message
            byte[] messageLength = Utils.integerToBytes((int) message.length);
            containerBytes = encodeBytes(containerBytes, messageLength,
                numBytesPerSample, 0);

            // Encode message
            containerBytes = encodeBytes(containerBytes, message, numBytesPerSample
                , 32);

            // Output
            outputStream.write(containerBytes);

        } catch (Exception ex) {
```

```
87              System.out.println("Ex2 :" + ex.getMessage());
                ex.printStackTrace();
89          } finally {
                try {
91                  if (containerStream != null) {
                        containerStream.close();
93                  }
                    if (outputStream != null) {
95                      outputStream.close();
                    }
97              } catch (Exception ex) {

99              }
            }
101
        }
103
        public static void decode(String inputFile, String outputFile, String password)
             {
105
            FileInputStream inputStream = null;
107         FileOutputStream outputStream = null;
            try {
109             // Create file streams
                inputStream = new FileInputStream(inputFile);
111             outputStream = new FileOutputStream(outputFile);

113             // Copy the header to the output file
                byte[] wavHeader = new byte[32];
115             inputStream.read(wavHeader);

117             // Get bits per sample
                byte[] bitsPerSample = new byte[2];
119             inputStream.read(bitsPerSample);

121             wavHeader = new byte[10];
                inputStream.read(wavHeader);
123
                // Read remainder of container
125             int bytesToRead = (int) new File(inputFile).length() - WAV_HEADER_SIZE;
                byte[] hiddenBytes = new byte[bytesToRead];
127             inputStream.read(hiddenBytes, 0, hiddenBytes.length);

129             int numBitsPerSample = Utils.bytesToInteger(bitsPerSample);
                int numBytesPerSample = numBitsPerSample / 8;
131
                // Decode message size
133             int length = 0;
                int offset = 32;
135             for (int i = 0; i < 32; i++ ) {
                    int pos = ((i + 1) * numBytesPerSample) ;//- 1;
137                 length = (length << 1) | (hiddenBytes[pos] & 1);
                }
139
                // Create array for containing message
141             byte[] message = new byte[length];

143             // Iterate bytes of message
                for (int i = 0; i < length; ++i) {
145                 // Iterate each bit of byte
                    for (int j = 0; j <8; ++j, ++offset) {
147                     int pos = ((offset + 1) * numBytesPerSample) ;//- 1;
                        message[i] = (byte)((message[i] << 1) | (hiddenBytes[pos] & 1))
                            ;
149                 }
                }
151
```

```
            try {
153             if (!password.isEmpty()) {
                    message = AES.decrypt(message, password);
155             }
                FileOutputStream fos = new FileOutputStream(outputFile);
157             fos.write(message,0,message.length);
                fos.close();
159         } catch (Exception ex) {
                System.out.print("An error occurred attempting to decode your
                    message. Invalid password?");
161         }

163     } catch (Exception ex) {
            System.out.println("Ex2 :" + ex.getMessage());
165         ex.printStackTrace();
        } finally {
167         try {
                if (inputStream != null) {
169                 inputStream.close();
                }
171             if (outputStream != null) {
                    outputStream.close();
173             }
            } catch (Exception ex) {
175
            }
177     }

179 }

181 public static String inspect(String containerFile) {
        return Utils.humanReadableSize(inspectBytes(containerFile));
183 }

185 public static long inspectBytes(String containerFile) {
        try {
187         long fileLength = new File(containerFile).length();


189
            FileInputStream containerStream = new FileInputStream(containerFile);
191
            // Get bits per sample
193         byte[] bitsPerSample = new byte[2];
            containerStream.skip(32);
195         containerStream.read(bitsPerSample);
            int bytesPerSample = Utils.bytesToInteger(bitsPerSample);
197
            int byteCapacity = (((int) fileLength) - WAV_HEADER_SIZE) /
                bytesPerSample;
199         // Remove 4 bytes needed to indicate message size
            byteCapacity -= 4;
201
            return byteCapacity;
203
        } catch (Exception ex) {
205         ex.printStackTrace();
            return 0;
207     }
    }
209 }
```

Listing A.2: Java version of the audio steganography tool

A.2 Image Steganography

In addition to an audio steganography tool we also produced an image steganography tool that is capable of taking any popular image format and encoding data into the LSB of pixel colours. Our tool only supports outputting of encoded images to the PNG palette-based format. The source code for this tool is provided in code listing A.3.

```
1  package us.steganosaur.steganography;
2
3  import us.steganosaur.cryptography.AES;
4  import us.steganosaur.utils.FileUtils;
5  import us.steganosaur.utils.Utils;
6
7  import javax.imageio.ImageIO;
8  import java.awt.image.BufferedImage;
9  import java.awt.image.DataBufferByte;
10 import java.io.*;
11 import java.awt.Graphics2D;
12
13 public class ImageSteg {
14
15     private static byte[] encodeBytes (byte[] container, byte[] toEncode, int
           offset) {
16         // Check that the message is not too long for the container
17         if ((toEncode.length + offset) > container.length) {
18             new Throwable(new Exception("Message too long for container"));
19         }
20         for (int i = 0; i < toEncode.length; i++) {
21             int byteValue = toEncode[i];
22             // Loop through each bit of the byte
23             for (int j = 7; j >= 0; j--,++offset) {
24                 // Get the bit value
25                 int bit = (byteValue >> j) & 1;
26                 // Set LSB to bit
27                 container[offset] = (byte) ((container[offset] & 0xFE) | bit);
28             }
29         }
30         return container;
31     }
32
33     public static void encode(String inputFile, String containerFile, String
           outputFile, String password) {
34
35         // Load input message, encrypt and convert to bytes.
36         byte[] message = null;
37         try {
38             message = FileUtils.getBytes(inputFile);
39             if (!password.isEmpty()) {
40                 message = AES.encrypt(message,password);
41             }
42         } catch (Exception ex) {
43             System.out.println("Ex1: " + ex.getMessage());
44         }
45
46         // Load containerFile into BufferImage
47         BufferedImage container = null;
48         try {
49             BufferedImage containerImage = ImageIO.read(new File(containerFile));
50             container = new BufferedImage(containerImage.getWidth(), containerImage
                   .getHeight(), BufferedImage.TYPE_3BYTE_BGR);
51             Graphics2D  graphics = container.createGraphics();
52             graphics.drawRenderedImage(containerImage, null);
53             graphics.dispose();
54         } catch (Exception ex) {
55             System.out.println("Ex2: " + ex.getMessage());
56         }
```

```
57
        // Container file as bytes
59      byte[] containerBytes = Utils.bufferedImageToBytes(container);
        // Message length as bytes
61      byte[] messageLength = Utils.integerToBytes((int) message.length);

63      encodeBytes(containerBytes, messageLength, 0);
        encodeBytes(containerBytes, message, 32);
65
        try {
67          ImageIO.write(container,"png",new File(outputFile));
        } catch (Exception ex) {
69          System.out.println("Ex3: " + ex.getMessage());
        }
71
    }
73
    public static void decode(String inputFile, String outputFile, String password)
         {
75
        // Load containerFile into BufferImage
77      BufferedImage container = null;
        try {
79          System.out.println(inputFile);
            BufferedImage containerImage = ImageIO.read(new File(inputFile));
81          container = new BufferedImage(containerImage.getWidth(), containerImage
                .getHeight(), BufferedImage.TYPE_3BYTE_BGR);
            Graphics2D  graphics = container.createGraphics();
83          graphics.drawRenderedImage(containerImage, null);
            graphics.dispose();
85      } catch (Exception ex) {
            System.out.println("Ex1: " + ex.getMessage());
87      }

89      // Container file as bytes
        byte[] hiddenBytes = Utils.bufferedImageToBytes(container);
91
        // Get message length
93      int length = 0;
        int offset = 32;
95      for (int i = 0; i < 32; i++ ) {
            length = (length << 1) | (hiddenBytes[i] & 1);
97      }

99      // Create array for containing message
        byte[] message = new byte[length];
101
        // Iterate bytes of message
103     for (int i = 0; i < length; ++i) {
            // Iterate each bit of byte
105         for (int j = 0; j <8; ++j, ++offset) {
                message[i] = (byte)((message[i] << 1) | (hiddenBytes[offset] & 1));
107         }
        }
109
        try {
111         if (!password.isEmpty()) {
                message = AES.decrypt(message, password);
113         }
            FileOutputStream fos = new FileOutputStream(outputFile);
115         fos.write(message,0,message.length);
            fos.close();
117     } catch (Exception ex) {
            System.out.print("An error occurred attempting to decode your message.
                Invalid password?");
119     }
```

```
121     }
123     public static String inspect(String containerFile) {
            return Utils.humanReadableSize(inspectBytes(containerFile));
125     }
127     public static long inspectBytes(String containerFile) {
            try {
129
                BufferedImage container = null;
131             BufferedImage containerImage = ImageIO.read(new File(containerFile));
                container = new BufferedImage(containerImage.getWidth(), containerImage
                    .getHeight(), BufferedImage.TYPE_3BYTE_BGR);
133             Graphics2D  graphics = container.createGraphics();
                graphics.drawRenderedImage(containerImage, null);
135             graphics.dispose();

137             // Container file as bytes
                byte[] containerBytes = Utils.bufferedImageToBytes(container);
139             int byteCapacity = (containerBytes.length - 4) / 8;

141             return byteCapacity;

143         } catch (Exception ex) {
                ex.printStackTrace();
145             return 0;
            }
147     }

149 }
```

Listing A.3: Java version of the audio steganography tool

Appendix B

Steganography System

B.1 Java System

The figures in this section show the level of functionality implemented in the GUI of the original Java application. These screenshots are not comprehensive of the full level of functionality of the system – some functions were not accessible via the GUI (see section 4.2.1 for further details).

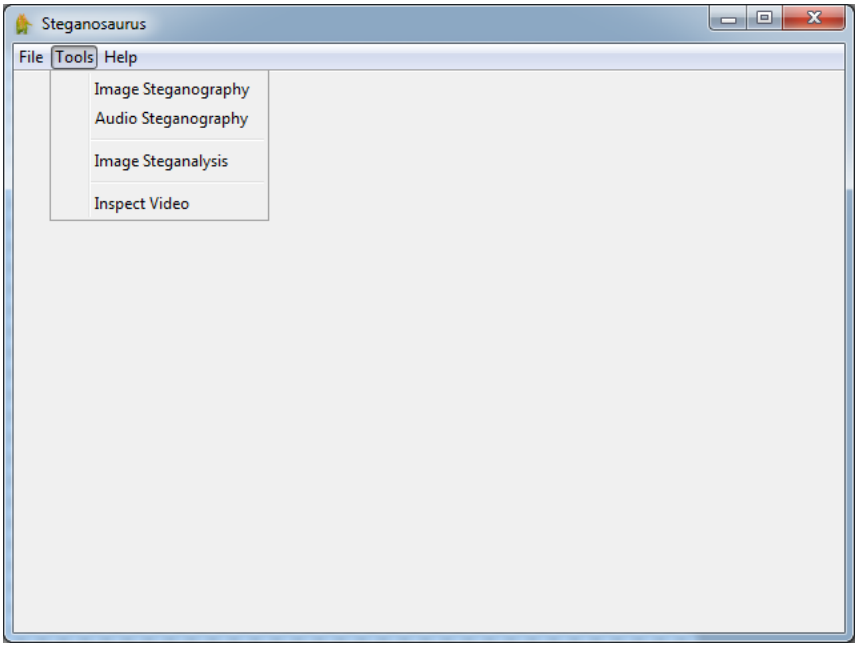

Figure B.1: Main interface

From the main interface, the following dialogs are accessible:

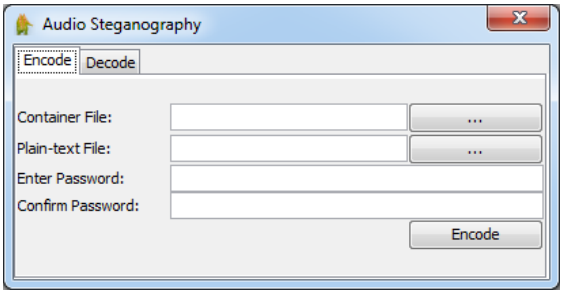

Figure B.2: Audio steganography tool - encoding mode

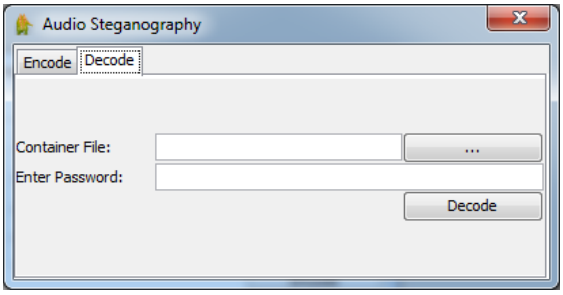

Figure B.3: Audio steganography tool - decoding mode

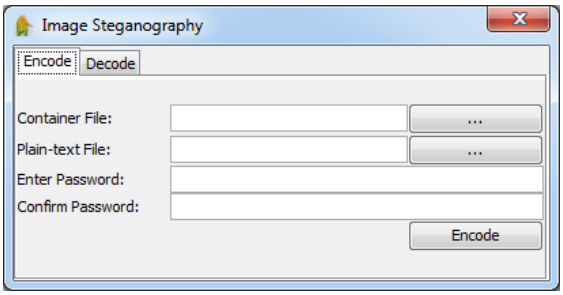

Figure B.4: Image steganography tool - encoding mode

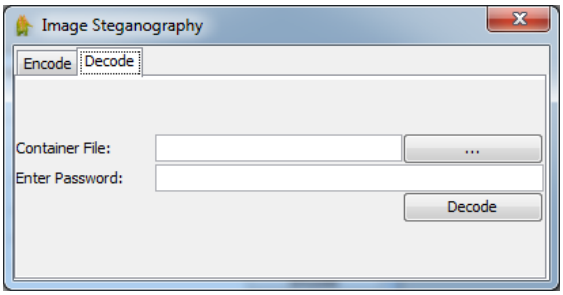

Figure B.5: Image steganography tool - decoding mode

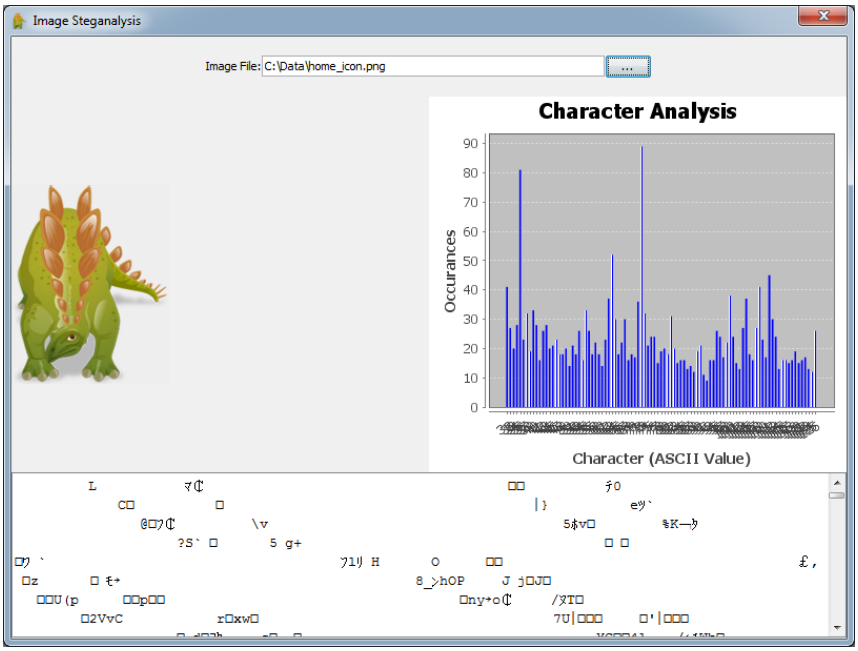

Figure B.6: Steganalysais screen showing when analysing an image *without* data embedded. The textbox contains the LSB string.

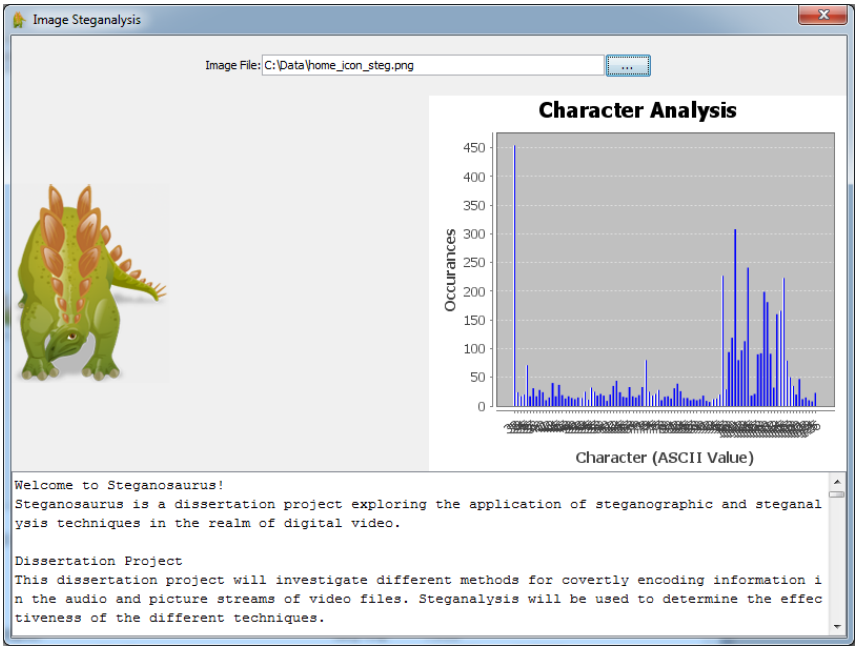

Figure B.7: Steganalysais screen showing when analysing an image *with* data embedded. The textbox contains the LSB string.

B.2 Makefile

Make is a utility that automatically compiles source code by reading a *makefile*. The makefile contains a set of rules that will derive the target program upon execution.

Code listing B.1 shows our solution for compiling several C files to work with the FFmpeg API. This code listing will work under the following prerequisites:

1. You are using Linux (preferably Ubuntu 12.10 LTS)
2. You have followed the FFmpeg compile guide for Ubuntu:
   `http://ffmpeg.org/trac/ffmpeg/wiki/UbuntuCompilationGuide`

```
CC = gcc
OBJECTS = general.o player.o transcoder.o
MAIN_OBJECTS = main.o ${OBJECTS}
INCLUDES = $(shell pkg-config --cflags libavformat libavcodec libavfilter
    libswscale libavutil sdl)
CFLAGS = -Wall -ggdb
LDFLAGS = $(shell pkg-config --libs libavformat libavcodec libavfilter libswscale
    libavutil sdl) -lm
EXE = steganosaurus.out

#
# $< is the first dependency in the dependency list
# $@ is the target name
#
all: $(EXE)

$(EXE) : $(MAIN_OBJECTS)
  $(CC) $(CFLAGS) $(OBJECTS) $< $(LDFLAGS) -o $@

%.o : %.c
  $(CC) $(CFLAGS) $< $(INCLUDES) -c -o $@
```

```
21 clean:
    rm -f *.o
23  rm -f $(EXE)
```

Listing B.1: FFmpeg makefile

Appendix C

steganosaur.us

Steganosaurus is the nickanme given to this project. From the offset a website (`http://www.steganosaur.us` was setup to document the research and developments of this project. Through the blog we recorded all progress, set backs, discoveries and additional details as frequently as possible.

Figure C.1: steganosaur.us - Homepage

The *Tools* section of our website allows visitors to use some of the steganography and cryptography tools that we produced during the preliminary research phase of this project.

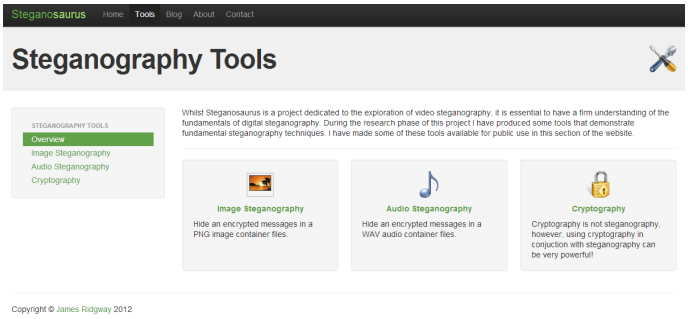

Figure C.2: Steganosaur.us - Tools section

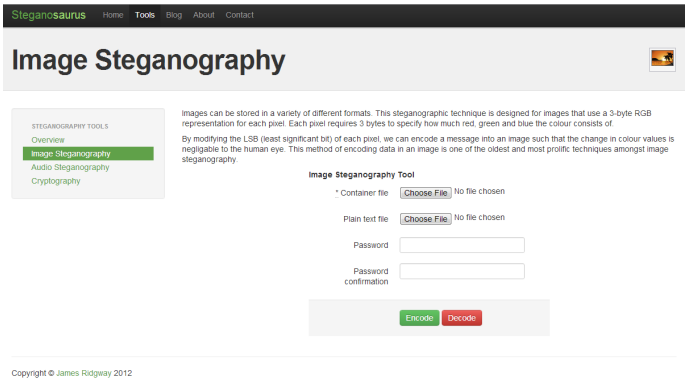

Figure C.3: Steganosaur.us - Audio steganography tool

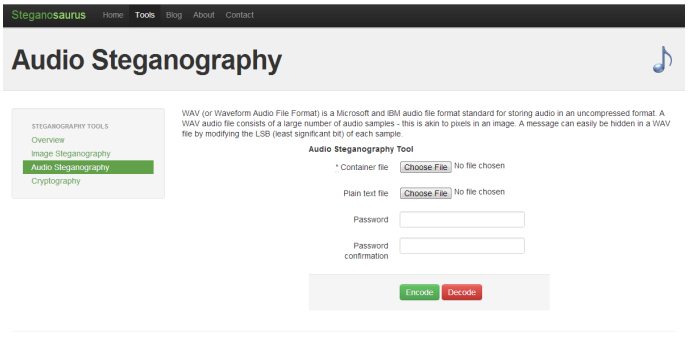

Figure C.4: Steganosaur.us - Image steganography tool

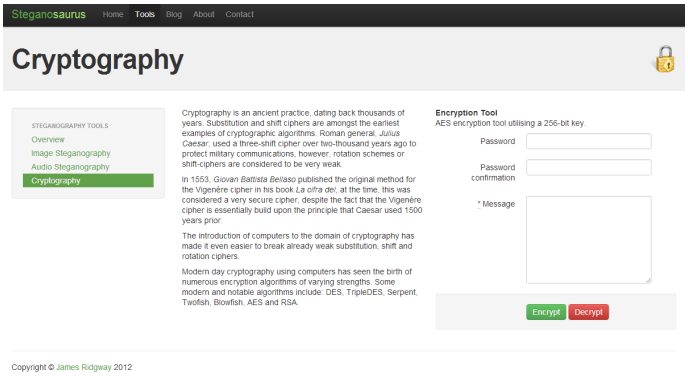

Figure C.5: Steganosaur.us - Cryptography tool