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

30th April 2013

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: 30th April 2013

Abstract

The field of digital steganography centres on hiding information in digital file formats. Whilst the application of steganographic techniques in relation to image and audio files has been extensively researched, research into the use of other container files remains limited.

The aim of this project is to explore different methods for securely encoding messages in a multimedia container, utilising both the audio and video stream, and using steganalysis to determine their effectiveness.

Acknowledgements

I would like to thank my supervisor, Mike Stannett, for allowing me to undertake this project, and for his unwavering support and guidance throughout. I would also like to thank:

- Michael Niedermayer for helping me navigate and familiarise myself with the vast FFmpeg codebase.
- My parents for their support in everything I do.
- My brother, Paul, for providing advice and being an unrivalled source of inspiration.

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.

During the early stages of this project I thoroughly researched the fundamentals of steganography, in addition to this I investigated some methods for video manipulation. My findings and the resulting products are documented in Appendix A. The research documented in Appendix A bears relevance to the main body of this document and is cross referenced accordingly.

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

Towards the end of this project I was asked to provide a special guest lecture for the Department of Computer Science. My lecture started with the history of steganography and covered video coding, audio, image and video steganography techniques, including specifics of this project. This lecture was pitched at:

- COM1004 Web and Internet Technology (Level 1) students.
- COM3501/4501/6501 Computer Security and Forensics (Level 3/Level 4/MSc) students.
- Departmental academics and researchers.

More details about the lecture can be found at: `http://www.steganosaur.us/lecture`

Conventions Used in This Document

The following typographical conventions are used in this document:

Italic
: Indicates a new term.

`Constant Width`
: Denote URLs and programming code (code snippets, class names, function names, etc.)

Contents

List of Figures

List of Tables

Glossary

Active warden	A warden that attempts to prevent the use of steganography by altering the message whilst preserving the meaning.
Alice	The name of a fictitious person in the Prisoners' Problem.
Audio stream	The audio component/track of a video file.
B-Frame	A bi-directional frame whose representation is predicted by its two neighbouring frames.
Blind steganalysis	Steganalysis where the attacker has no knowledge of the stegosystem.
Bob	The name of a fictitious person in the Prisoners' Problem.
Ciphertext	The output of bits produced by a cryptographic algorithm.
Container	In steganography a container is the object in which a message is hidden.
Cryptography	The practice and study of secure communication.
Cryptosystem	A system of algorithms that is capable of performing encryption and decryption tasks.
Forensic steganalysis	Forensic steganalysis is a level of steganalysis that attempts to determine the nature (message contents, length etc.) of a message hidden in a container.
Frame	A raw (decoded) portion of data that contains either audio samples or an image.
Generation	A steganographic technique that generates the container file based on the information to be encoded.
I-Frame	Shorthand for an Intra Frame.
Injection	A steganographic technique that inserts additional information into a container file.
Intra Frame	A reference frame of a video that is independent from any other frames.

Kerckhoff's principle	The principle that a cryptosystem should remain secure if everything except the key is known.
Linguistic steganography	A form of steganography where the covert message is contained in text.
Malicious warden	A warden that attempts to capture the communication of parties by impersonation and/or extensive modification of messages.
P-Frame	A predicted frame whose representation is predicted from preceding frames.
Packet	A packet is a compressed and formatted unit of data.
Passive warden	A warden that passively observes communication.
Picture stream	The image component of a video file.
Plaintext	Message to be encrypted by a cryptographic algorithm.
Prisoners' Problem	A fictitious scenario used to illustrate the difficulties of steganography and steganalysis.
Spatial domain	The spatial domain refers to the normal space of a multidimensional signal. In the context of an image, this is the 2D pixel space.
Spatiotemporal domain	Spatial domain signals existing in the temporal domain.
Statistical steganalysis	The process of performing steganalysis by representing the container as a set of numerical functions.
Steganalysis	The art and science of detecting messages that have been hidden using steganography.
Steganographic capacity	The amount of information that can be hidden in a container file.
Steganography	The art and science of covert communication.
Stegosystem	A covert system for communicating that comprises a container file, steganographic key and message. The stegosystem also encompasses the physical exchange of messages.
Substitution	A steganographic technique whereby information in the container file is replaced by the message contents.

Targeted steganalysis	A type of steganalysis attack that utilises information about the steganographic algorithm or stegosystem.
Temporal cohesion	The correlation between signals observed at different moments in time.
Transform domain	A mathematical procedure for converting data from one domain to another.
Warden	A person or computer program that monitors the communication between parties (such as Alice and Bob in the Prisoners' Problem).
Watermarking	The technique of embedding supplementary data in a container file.
Wendy	The name of the fictitious warden in the Prisoners' Problem.

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 17th Century, when Johannes Trithemius uses the word in a trilogy published in Frankfurt in 1606. The first two volumes, *Polygraphia* and *Steganographia* specifically discuss *cryptography* and steganography [Fri10, Col03].

1.1 Project Aims

The aim of our project was to produce a cross-platform video steganography tool that is capable of hiding data securely in a video *container*. Video manipulation is a broad and complex topic, therefore we limited our research to a single video format (H.264/AAC – a commonly used format for recording high-definition video). Verifying the security of any system or solution is not a simple process done over a couple of weeks or months. Whilst some understanding of the security of the system can be achieved in this time frame, many security experts believe a system is only secure if it has not been broken after several years [Col03]. Our final software solution combines steganography and cryptography to guarantee a minimum level of security.

1.2 The History of Steganography

1.2.1 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 [Whi90].

1.2.2 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*

This type of linguistic steganography is sometimes called a *null cipher* [Rab04].

1.3 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.

1.3.1 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].

Two individuals (*Alice* and *Bob*) are imprisoned. Alice and Bob can communicate with each other, but all of their communication is constantly monitored by a *warden* (*Wendy*). Alice and Bob want to hatch an escape plan, to do this they will use steganography to communicate covertly. It is crucial that their communication is undetectable, since the mere presence of a secret message will alert Wendy.

1.3.2 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 the presence of a covert message is detected 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.

1.3.3 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].

1.4 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 Prisoners' Problem, steganalysis is the job of the warden. Wardens can be associated with a variety of categories such as: active, passive and malicious.

1.4.1 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].

1.4.2 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.

1.4.3 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].

1.5 Video Steganography

Digital steganography techniques often use image and audio files as carriers for their hidden information. Between 1992 and 2007, images were by far the most popular type of steganographic container with 52% of steganography software supporting image containers. Although audio files are the second most commonly supported format, only 15% of steganography software uses audio files. The use of video files as steganographic containers is still a developing area of the field, with only 3% of steganography software supporting these files [DHC10].

1.6 Structure of this Report

The remainder of this report is divided into six chapters. In Section 2 we review the existing literature starting with an overview of the fundamental principles and techniques, before discussing steganography and steganalysis in detail. In Section 3 we analyse the requirements and goals of this project, and identify how the success of the project will be evaluated. Section 4 documents the designs and implementation of our system, and the agile design process used to develop our solution. Section 5 details the testing that has been carried out on our solution. In Section 6 we present the results of our work, the objectives that we have achieved and discuss further research that could be carried out within the field of video steganography. Finally, in Section 7, we summarise our findings

and the points discussed in this report.

Chapter 2: Literature Survey

There are numerous techniques for hiding data in a digital container file. Although this project focuses 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*, *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 End of File (EOF) 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 WAV. 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.2.1), 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 substitution) 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 Techniques

A substitution 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.

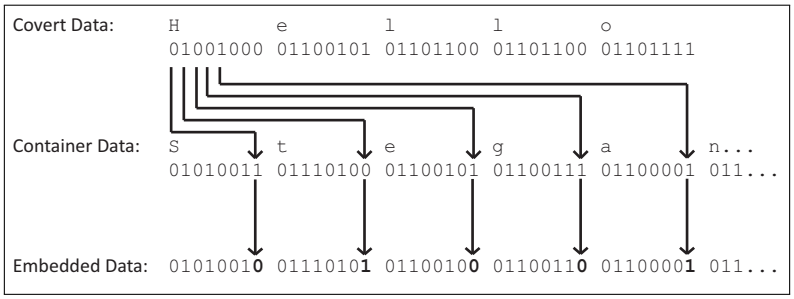


Figure 2.1: This figure shows LSB encoding of the word "Hello" inside container data.

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 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-dimensional 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, which makes LSB manipulation redundant when working 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 Section 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 losslessly 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 over other steganographic techniques that required an existing input container. If the original (unmodified) 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 disadvantaged by the fact that producing a steganography system that generates realistic fractals is complex 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, ÓDB96]. 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 2.4). 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 discrete cosine coefficients.

Discrete Cosine Transform

The DCT 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 or 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 Transform (DWT) [KK10].

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

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

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

where

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

and $f(x, y)$ is the intensity of the pixel at the x^{th} row and y^{th} column.

As mentioned previously, 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 high-definition video) container files are significantly larger than audio or images files, thus reducing the problem of steganographic capacity. Noda et al. overcame steganographic capacity issues by using a *Bit Plane Complexity Segmentation* (BPCS) technique for wavelet compressed video data [NFNK04]. Similarly, Jalab et al. propose 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 2.4). 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 A.8 and Section A.3.1) [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 note, 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 analysing 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].

Motion Vector Techniques

The techniques described in this section focus on embedding data in motion vectors – an explanation of motion vectors can be found in Section 2.3.1. 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 Advanced Encryption Standard (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 quantisation 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 quantisation scales. The number of quantisation 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 quantisation 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 essential, 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 Xu et al. [XPZ06]: his findings show a better balance of the payload (covert data) between P- and B- frames with his method [Aly11].

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.

2.2.2 Streaming and Real Time

Streaming videos across the internet has become an incredibly popular activity over the last ten years [Mar03]. 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 substitution-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 [LLLL06] it is a fragile technique that neglects the resilience aspect of steganography. The strengths, weaknesses and approaches of the methods proposed by

[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 [Con08].

Mozo et al. and Liu et al. vary significantly. Injection-based methods (such as Mozo et al.) are 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, no form of data transmission, including internet streaming is 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.'s approach is disadvantaged.

2.3 Video Coding

In this section of the literature survey we will explore the structure and format of video files, focusing on aspects of video which are relevant to the H.264 standard. This section starts with a more in-depth analysis of compression as used in multimedia formats before addressing the specifics of video files.

2.3.1 Compression

So far we have looked briefly at how DCT is used to compress an image. Multimedia formats, whether they be image, audio or video, can use compression techniques to reduce the size of data that is needed to represent the original media. Most compression techniques rely on producing an "alternative representation" in a domain different to that of the original media [Muk11].

The Compressed Domain

Images are originally represented in the spatial domain, which as a function is represented as $I : \mathbb{Z}^2 \rightarrow \mathbb{N}^3$ whereby each coordinate of a 2D coordinate space maps to a three-dimensional RGB colour vector. Video, being sequences of images is represented in the *spatiotemporal domain* such that $V : \mathbb{N} \times \mathbb{Z}^2 \rightarrow \mathbb{N}^3$ [Muk11, Fri10].

Let's consider the implications of storing the output of these functions in an uncompressed format. If we were to consider a single 1920 x 1080 image just over 2 million pixels values will need to be stored. This is a substantial amount of information to store for a single image. In the context of a standard video file[3], a single second of video at 1920 x 1080 resolution would require between 49 and 63 million pixel values. Images and videos contain vast quantities of data and storing this data in a raw, uncompressed format is impractical for most situations, as a result, image and video formats typically use an alternative representation. DCT and DWT are two of the more popular methods for alternatively representing images and video, both of which belong to the compressed domain [Muk11, Fri10].

Compression Considerations

Before data is represented in a compressed format important consideration should be given to the trade-off between computational cost and storage requirements.

[3]Most videos range between 24 and 30 frames per second.

The accuracy of the representation is another important consideration. For images and video, an approximation of the original source if often sufficient, which means that lossy compression schemes can be used, however, if the given application requires an exact representation then a lossless representation must be used [Muk11].

Image and video compression methods should consider the following attributes [Muk11]:

- **Reconstructibility** From an encoded (compressed) representation it should be possible to decode/reconstruct the original data. In the case of lossy compression the reconstruction can be an approximation.
- **Low Redundancy** The compressed representation should be as concise as possible. Images in the spatial domain can have a high redundancy rate due to high spatial correlation of neighbouring pixels. Correlation can also exist in the spatial domain if we consider the context of video, neighbouring frames will have high temporal correlation because the change in an image frame is very gradual from one consecutive frame to another.
- **Factorisation into substructures** Decomposing an object into its component parts can be useful for identifying those components whose contribution is relatively insignificant. Insignificant parts can be removed or reduced further, thus reducing the size of the compressed representation.

In Section 2.1.4 we saw how coefficient-based transforms such as DCT, DWT and even a Discrete Fourier Transform (DFT) [Mit00] can be used to represent an image by the form of an *alternative representation*. Whilst a single video frame can be represented independently using any image representation technique, these methods do not harness the *temporal cohesion* [Muk11]. Consecutive frames can have high temporal redundancy because portions of the frame remain unchanged, and by examining the correlation between consecutive frames a more concise representation can be used [Muk11, Ric08, Le 91].

A better representation of videos frames with a reduced temporal redundancy is to use a Group of Pictures (GOP) technique. In a GOP, one of these frames will be a reference frame which is called an *Intra Frame* or *I-Frame*. Other frames are then predicted by this, and these predictions are represented as changes (deltas) from the preceding frame – these are known as *P-Frames*. Some predicted frames are predicted from its two neighbours, these are bi-directionally predicted frames and are called *B-Frames* [Muk11, Ric08].

Prediction errors and motion vectors are used to explicitly represent temporal cohesion, however, there are some instances where this is not the case. Some frames are represented as side information or parity bits that can be used decode the frame – these frames are only approximations. Side information and parity bit encoding is often used for low complexity compilers or in distributed environments so that each frame is encoded individually – this method of encoding is called *distributed video coding* [Muk11, GARR05]

2.3.2 Quantisation

Quantisation is an important part of lossy compression, and accounts for its "lossy"-ness. Compressed data is a smaller alternative representation of the original data. This process of converting from one representation to another can often involve an intermediate representation. Quantisation is the process of scaling down the range of symbols that are used in the intermediate representation. For instance, with a DCT a matrix of coefficients is

produced whose values may range between −223 and 150, but after quantisation, these values may only range between 10 and 130. The reduced range caused by the quantisation process subsequently requires fewer bits to code the representation than the original range. Quantisation parameters for multimedia formats are chosen based on how individual components affect the average human perception [Muk11].

2.3.3 Coding Concepts [4]

An encoder (compressor) and decoder (decompressor) forming a complementary pair is known as a *codec* (enCOder/DECoder). The encoder is used to store or transmit video by converting the original raw video format to an alternative (compressed) representation. The decoder converts the compressed form back to the original video.

The H.264 codec consists of four main components: block-based motion compensation, transform, quantisation and entropy coding.

A codec uses a model (an efficient alternative representation) to reconstruct an approximation of the original video files. A codec should attempt to maximise on two conflicting goals: high fidelity (high quality) video with high compression levels. Decoding an alternative video representation that uses few bits often results in the decoder outputting a poor, low quality approximation.

A video encoder consists of three core components: a *temporal model*, a *spatial model* and an *entropy encoder*. The temporal model reads in a sequence of video frames and attempts to reduce the temporal redundancy by identifying similarities between the neighbouring video frames – this analysis usually involves computing a prediction of the current video frame. With H.264 the prediction can be computed from multiple previous or future frames. The prediction is improved by means of compensation for differences between the frames – this is known as motion compensation prediction. The temporal model outputs a residual frame and a set of motion vectors. The residual frame is computed by subtracting the prediction from the current frame, motion vectors are used to describe how the motion was compensated.

The residual frame from the temporal model is then fed into the input of the spatial model. The spatial model, like the temporal model is concerned with removing redundancy in its domain, in this case, the spatial model analyses neighbouring samples of the residual frame (produced by the temporal model) to reduce the spatial redundancy. Spatial reduction in H.264 is achieved by applying a transform followed by a quantisation process. The transform step produces a set of transform coefficients which are then quantised, removing insignificant values, and returning the quantised transform coefficients as the output of the spatial model.

The entropy encoder is the final component of the video encoder that produces an encoded output from the results of the spatial and temporal model. The entropy encoder processes the motion vectors from the temporal model and the coefficients from the spatial model to produce a compressed bit stream consisting of motion vectors, residual transform coefficients and header information.

Although the quantisation stages cause a loss of information, this process is roughly reversible, and as such, the decoder mechanism works in reverse to that of the encoder. The output produced by the decoder mechanism will only ever (in the case of H.264) be an approximation to the original input because of the quantisation stages.

[4] Unless explicitly stated otherwise information in this section is based on [Ric08] and [Muk11].

Temporal Model

The residual frame produced by the temporal model is produced by subtracting the predicted frame from the actual video frame. The resize of the residual frame is dependent on the accuracy of the prediction process – the smaller the residual frame, the fewer bits needed to code it. Prediction accuracy can be improved by calculating and propagating compensation for motion from the reference frame(s) through to the current frame.

Motion compensation can significantly improve prediction calculations because two successive video frames are usually highly correlated. Most of the information captured in successive residual frames relates to the movement of objects in the scene, therefore a better prediction can be produced by compensating for this change in motion between frames. Changes between video frames can be caused by the motion of objects, changes in the scenery and adjustments in light levels. With the exception of lighting and changes to the scenery, these changes directly correspond to the movement of pixels between frames. The movement of individual pixels between successive frames can be estimated. The displacement movement of pixels in a video frame is known as an *optical flow* [AWSZ05].

In theory, it is possible to use the optical flow to predict the majority of the pixels in the current frame, provided the optical flow is accurate, by displacing each pixel as described by the optical flow.

Unfortunately, this is a very computationally intensive process, as each pixel will have to be transformed, and each frame decoded, on a pixel-by-pixel basis using the optical flow vectors. Whilst workable in theory, this would result in a large amount of residual data, which is at odds with the desirability of a compact residual frame.

Macroblocks Motion Estimation

A macroblock is typically a 16×16 pixel block of the current frame, in the wider context of block-based motion estimation a block is any $N \times M$ sample of a frame. Macroblocks are used by a variety of codecs including MPEG-1, MPEG-2, H.261, H.263 and H.264.

Macroblock motion estimation starts by dividing each frame into macroblocks. In turn, each macroblock is taken, and the reference frame is searched for a matching macroblock. Macroblocks from the current frame are paired with macroblocks in the current frame by choosing a candidate block that minimises the difference between the macroblock in the current frame and itself – this process provides a residual block. Finally, the residual block is encoded and stored, alongside the difference between the current macroblock and the candidate macroblock, called a motion vector.

Using a 16×16 size macroblock can cause some problems with certain motions and object outlines. If a macroblock and its corresponding candidate macroblock have a large difference (residual energy), then the number of bits required to code this macroblock increases and inflates the bit-rate. This issue can be addressed rather simply by decomposing a macroblock into smaller 8×8, or even 4×4 macroblock size. Using smaller macroblocks results in a larger number of blocks, which can be disadvantageous, therefore a better solution is to use an adaptive block size approach as used by H.264.

2.4 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 1.3.1)

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].

2.4.1 Overview of Steganalysis Techniques

Steganalysis techniques can be split into two different categories:

- *Targeted steganalysis*
- *Blind steganalysis*

Both targeted steganalysis 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].

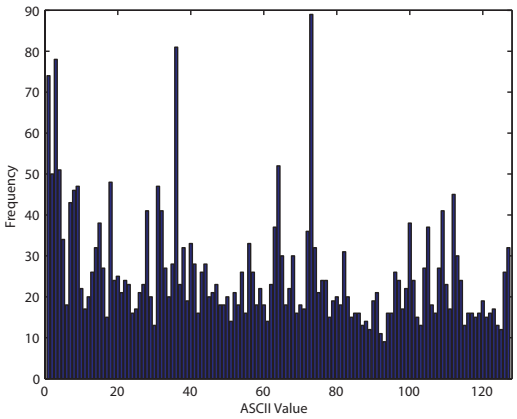


Figure 2.2: ASCII distribution from LSB string before data is embedded in a PG image.

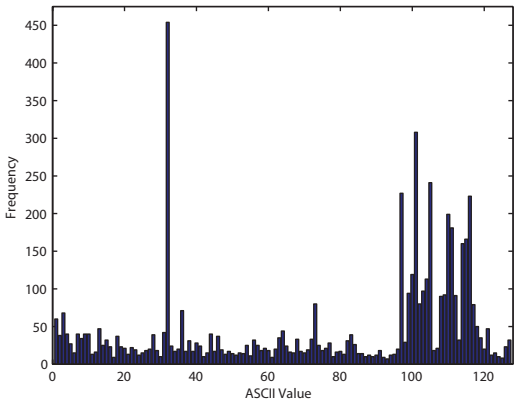


Figure 2.3: ASCII distribution from LSB string after data is embedded in a PG image.

Figure 2.4: Comparison of ASCII distributions from LSB strings of image data

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 significantly 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 steganalysisblind and targeted steganalysis.

2.4.2 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

Example:
```
Plaintext:  the quick brown fox jumps over the lazy dog
Ciphertext:  wkh txlfn eurzq ira mxpsv ryhu wkh odcb grj
```

Figure 2.5: Caesar cipher example

a frame [BKZ06, ZSZ08, PDB09, JKH07]. Steganographic approaches that utilise 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].

2.5 Cryptography

Cryptography is the study and practice of secret writing. Evidence of cryptography can be found in the earliest forms of writing and date back thousands of years. Despite the age of the field, improvements and innovations have not been frequent with many schemes being invented and reinvented since its inception [Dav97].

2.5.1 Substitution Ciphers

Caesar's cipher is one of the oldest and most well-known substitution ciphers. Encrypting a message with the Caesar cipher can be done by replacing each character with the character that occurs 3 to the right in the alphabet (modulo 26) [Sch96] – see figure 2.5.

Substitution schemes dominated the field of cryptography between 1400 and 1850. Most schemes like the Caesar cipher, used a fixed table of letter substitutions. Substitution ciphers are very weak because they do not adjust the letter frequencies. In the English language "E" is the most common letter, so the most frequent letter of a fixed substitution cipher is likely to be E (assuming the original message is written in English) [Dav97,Sch96].

The *Vigenère* cipher was another popular substitution based cipher that became popular in the 19^{th} century [Dav97]. This cipher is regarded as a polyalphabetic substitution because it involves multiple substitution alphabets [BF11]. The encryption and decryption process is described below mathematically, where C represents the *ciphertext*, M the *plaintext* and K the key. The size of the key is denoted by $|K|$.

Vigenère encryption:

$$C_i = E_k(M_i) = (M_i + K_{i \bmod |K|}) \bmod 26 \tag{2.3}$$

Vigenère decryption:

$$M_i = D_k(C_i) = (C_i - K_{i \bmod |K|}) \bmod 26 \tag{2.4}$$

In addition to polyalphabetic ciphers like the Vigenère, the *Playfair* cipher is an example of a polygram substitution cipher, whereby groups of letters are encrypted together. The Playfair cipher was invented in 1854 and used by the British during World War I [Sch96, Kah67].

2.5.2 Symmetric Algorithms

Symmetric encryption algorithms use a single cryptographic key (regarded as a shared secret) to perform encryption of plaintext and decryption of ciphertext [Sch96, DK07]. In most cases a symmetric encryption algorithm will use exactly the same key for encryption as it does for decryption, however, some symmetric algorithms work by deriving the encryption key from the decryption key, or vice versa [Sch96].

The security of symmetric algorithms ultimately rests with the security of the key. The key must remain secure for as long as the encrypted contents must remain secure [Sch96].

Symmetric algorithms, due to their design, are either *block* ciphers or *stream* ciphers. A block cipher will operate on groups of bits (referred to as a block). Most modern block ciphers have a block size of 64-bits. On the other hand, stream ciphers will operate on an individual bit-by-bit basis [Sch96].

Data Encryption Standard (DES)

Although currently regarded as insecure, Data Encryption Standard (DES) is one of the earliest and best known algorithms for encrypting electronic data. DES, a 56-bit block cipher, was developed by Horst Feistel as part of his work for IBM in the 1970s [Fei73, Smi71], by 1979 it would become standardised as the U.S Data Encryption Standard [Sch96]. DES remained as encryption standard for 18 years until January 1997.

Advanced Encryption Standard (AES)

The AES is the current de facto encryption algorithm for securing sensitive information, and is used by governments and militaries across the world.

In January 1997 the National Institute of Standards and Technology (NIST), a subdivision of the U.S Department of Commerce, start looking for a replacement to the DES which had been the encryption standard for nearly two decades.

Candidate algorithms were submitted to NIST for consideration as alternatives to DES. All of the algorithms submitted were subjected to review and analysis by NIST and the general public. The candidate algorithms initially submitted to NIST were: *CAST-256*, *CRYPTON*, *DEAL*, *DFC*, *E2*, *FROG*, *HPC*, *LOKI97*, *MAGENTA*, *MARS*, *RC6*, *Rijndael*, *SAFER+*, *Serpent*, and *Twofish*. These 15 algorithms were then narrowed down to a shortlist of: *MARS*, *RC6*, *Rijndael*, *Serpent*, and *Twofish* [Wan09].

Rijndael (pronounced "rain dahl") was finally chosen as the AES out of the shortlist of candidate algorithms. The Rijndael algorithm was developed by Belgian cryptographers, Joan Daemen and Vincent Rijmen.

The exact workings of AES and symmetric cipher block modes are discussed in considerable detail in Section B.

2.5.3 Asymmetric Algorithms

Asymmetric algorithms or public-key cryptography works on the premise of a public key and a private key. The public key can be known for anyone and is used to encrypt plaintext, the private key should remain secret to the owner and is used to decrypt ciphertext [Sch96, DK07]. In symmetric cryptography the encryption and decryption key are identical, or are derived from one another. Asymmetric cryptography survives on the premise that the private key cannot be easily derived in any reasonable amount of time from the public key [Sch96].

The history of public key cryptography in the literature is rather interesting. The initial idea of public and private key cryptography was first published by Diffie and Hellman in their paper, *New Directions Cryptography* [DH76] in 1976. Two years later Rivest, Shamir and Adleman published the RSA cryptosystem in [RSA78].

Prior to these publications James Ellis and Clifford Cocks, mathematicians and cryptographers working for the UK's Government Communication Headquarters (GCHQ) had already outlined concepts for asymmetric cryptography in private publications. In 1970, Ellis wrote his paper, *The possibility of non-secure digital encryption*, in which he provides the earliest practical mention of public key cryptography [Ell70]. The paper was kept as a private document until December 1997 when it was published alongside 4 other papers. Amongst those other papers was, "A note on non-secure encryption" – a paper written by Cocks in 1973 which for all intents and purposes describes what is now widely known as the RSA algorithm [Coc73].

RSA Cryptosystem

RSA utilises two large and distinct prime numbers p and q for the generation of public and private keys. The process of generating the keys is the most complex part of the RSA algorithm. The security of the RSA algorithm resides in the fact that prime factorisation of a number is a very expensive computational process. Depending on the size of the number, prime factorisation can take, hours, days, months or even years. In the RSA algorithm, the private and public key are generated as follows [FSK10, Sti02]:

1. Two distinct primes p and q are chosen at random.
2. Compute $n = pq$
3. Compute $\phi = (p-1)(q-1)$
4. Choose an integer e such that e and phi are coprime.
5. d is calculated as the inverse of e mod ϕ.

Encryption Process

To explain the encryption process let us assume that Alice wants to send Bob a message, M. Before Alice can send anything encrypted she needs Bob's public key that consists of n and e. Alice first converts the message M into an integer m, the corresponding ciphertext for this message is then calculated as follows:

$$c \equiv m^e \bmod n \tag{2.5}$$

Alice then sends c to Bob. This process of encrypting is not reasonably reversible, as it is computationally hard to derive m from x, e and n.

Decryption Process
Bob can decrypt Alice's ciphertext, c, by using his private key, d.

$$m \equiv c^d \bmod n \tag{2.6}$$

Encryption and Decryption Key
The public encryption key consists of the pair (e, n) and the decryption key is formed of the pair (d, n), additionally p, q and ϕ must also be kept secret.

The encryption and decryption processes shown by 2.5 and 2.6 are very similar. Despite their similarity the method remains secure because finding the prime factorisation of n is computationally hard. If an attacker was able to factorise n in a reasonable time they could determine p and q, hence ϕ, and hence d. Provided (d, n), p, q and ϕ remain secure it is not possible to derive d from e.

2.6 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 and 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.

Codecs generally encode data using information relative to other reference frames stored in the bit-stream. Although the structure of a video file may be complex and delicate, the process of encoding and decoding a video can be easily broken down into component models and entropy encoding. Motion estimation and motion prediction play a key role in reducing the bits required to encode a video frame. Motion estimation is commonly achieved using fixed size macroblocks, although some formats such as H.264 adopt an adjustable macroblock approach.

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.

The field of cryptography provides useful methods for securing messages and information. Straightforward substitution ciphers are now a thing of the past with widely adopted asymmetric and symmetric cryptographic algorithms being at the forefront of the field. AES and RSA are two highly secure and heavily utilised encryption algorithms.

This chapter has reviewed the various steganographic techniques, including some fundamentals that have been documented in the literature. The workings of the relevant

video formats and coding methods have also been researched. Finally, consideration has been given to the security of steganographic methods and how these can be fortified with the application of steganography.

Chapter 3: Requirements and Analysis

3.1 Project Overview

The primary aim of this project involved developing a cross-platform video steganography system that is capable of embedding data in a video container file using a motion-vector based approach. The intention was to produce a system that is both highly secure and practical. Once we established our initial embedding method our intention was to further develop other schemes and compare their effectiveness using steganalysis.

The requirements and analysis that are discussed below were adjusted as the project developed, due to the various design and implementation decisions outlined in Chapter 4 – this is a consequence of the agile methodology used during project development.

Measuring the security of a system is contentious as any security mechanism should have a timeless quality to it, whereby a system that is developed today should be secure for years to come. Irrespective of how comprehensive our steganalysis, testing and evaluation methods are we will not be able to determine whether our system is sufficiently secure in the timescale that this project permits. We have opted to address this issue by utilising cryptography within our steganographic solution.

By incorporating tried and tested cryptographic methods we aim to ensure at least a minimum level of security for the messages that are being communicated. We would like to stress that this will not inherently make the steganographic schemes we devise less detectable. However, as we saw in Section 2.2 of the literature survey, encryption can be used in conjunction with steganography to obfuscate the presence of embedded ASCII messages.

3.2 Requirements

In this section we outline the functional and non-functional requirements for our project. Both the functional and non-functional requirements are categorised into *mandatory*, *desirable* and *optional*. Mandatory requirements are necessary for our system to function correctly. Any desirable requirements are important, but, our system will still function without these being implemented. Optional requirements will provide additional enhancements to our system, but we should only consider implementing these if we have the time. Our goal will be to implement all of the mandatory and as many (but hopefully all) of the desirable features.

3.2.1 Functional Requirements

- Mandatory

1. Underlying transcoding mechanism – iterate and parse video frames.
2. Allow a steganographic scheme to modify the motion vectors of a video frame.
3. Retrieve and inspect video meta data to determine that the specified video file uses codecs supported by our system. Also allow the user to inspect details of a video file.
4. Implement a basic embedding algorithm that can use an LSB approach to embed one bit of data per frame.
5. Implement an extraction algorithm capable of retrieving messages encoded as per requirement 4.
6. Use AES cryptography to encrypt the message before it is embedded.
7. Use AES cryptography to decrypt the message after it has been extracted.
8. Steganalysis – 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.
9. Steganalysis – for a specified frame of the original and steganographic video file, allow the corresponding motion vectors to be analysed and compared.
10. Support the embedding of standard text-based ASCII messages.

- Desirable

11. Use different keys for steganography and cryptography.
12. Use a steganographic key to determine the placement of covert data in the container file.
13. Support the embedding of message data without constraints on the format of the data to be embedded.
14. Implement additional, more complex, embedding schemes.
15. Implement counterpart extraction schemes for any additionally implemented embedding schemes.
16. Automatically detect parameters (steganographic scheme, etc) when extracting an embedded message.
17. Steganalysis – allow motion vector analysis of neighbouring frames to spot uncharacteristic changes in motion vectors.

- Optional

18. Implement an embedding and extraction scheme that utilises the audio stream.
19. Allow the user to define parameters for steganographic schemes (e.g. low steganographic capacity and difficult to detect vs. high steganographic capacity and easy to detect).
20. Develop more sophisticated steganalysis tools.

3.2.2 Non-Functional Requirements

- Mandatory

  21. Cross-platform – supports Windows, Linux and Mac OS operating systems.
  22. The application should be easy to use.
  23. The graphical user interface should be clean, intuitive and unambiguous.
  24. Progress of time consuming processes such as embedding and extraction should be clearly indicated to the user.
  25. The system should support appropriate video formats.
  26. Allow the user to specify a steganographic/cryptographic key.

- Desirable

  27. The extraction process should be fast.
  28. The embedding process should be fast.
  29. The user should be able to select their preferred embedding scheme.
  30. Warn the user if the payload is greater than the steganographic capacity of the container before embedding begins.

- Optional

  31. Support multiple motion-vector based video codecs.
  32. Support automatic software updates.

3.3 Analysis

3.3.1 Application

The requirements define an application capable of embedding data using steganographic methods, and providing tools to analyse and compare video files for the purpose of steganalysis. Requirements 21, 22, 25, 32 relate specifically to the overall application, whereas requirements 23 and 24 detail the requirements for the graphical user interface of the application.

The aforementioned requirements focus on ease of use, clarity and portability. In [Sch96], 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 an ideal. Therefore, specific consideration has been given to the simplicity and easy of use of the application as a complex security tool will deter most users because of the inconvenience of complexity. Our requirements look at hiding the complexity of video steganography from the user, by the use of a clear graphical user interface (requirements 22 and 23).

3.3.2 Steganography

Applying steganographic techniques to a video file will rely on the ability to parse such a file (requirement 1) and demultiplex audio packets from video packets so that each video frame can be processed before multiplexing the resulting packet data for output as a video file.

For each video frame that is processed we will be required to inspect the motion vectors (requirement 2), and determine what modifications need to be made to the vector in order to embed information. Requirements 4 and 5 will utilise the ability to modify motion vector values for basic LSB embedding and extracting. The requirements stipulate a basic encoding and extracting method under the mandatory requirements because we were initially unsure of the complexities associated with working with video frames. Whilst several papers relating to the use of motion vector encoding are discussed in Section 2.2.1 of the literature survey, these papers merely provide a high level description of the technique without providing an insight into how to extract or modify motion vector values. These papers cover steganographic techniques for a variety of codecs, and as per requirement 25, we will select a motion vector based codec that is best suited to motion vector manipulation. Our research thus far indicates that the H.264 codec will work well for motion vector manipulation. Other macroblock based codecs such as H.261 and H.263 are somewhat restrictive as they support a limited range of video frame resolutions.

Requirement 10 states that the system only needs to support text-based ASCII messages initially – simplifying the data that we are processing should facilitate the construction of a stable steganographic system before introducing potential complexities that might arise from working with other data. Only supporting an ASCII text messages is a considerable limitation, therefore it is desirable that the system progresses to support any message (requirement 13). Implementing requirements such as requirement 13 will give greater control to the user and provide them with a greater level of flexibility in how they use the application.

Constraints on the type of data that can be embedded in a video container have arisen from the limitations that were discovered during the design and implementation phase (see Section 4).

As we have already identified in this report, the security of our steganographic system will be extremely difficult to ascertain, especially during the lifetime of the project, therefore the use of AES cryptography to secure the message will be mandatory (requirements 6 and 7) – at the very least this will ensure that the contents of the embedded message are secure. The user will be able to set a steganographic key (requirement 12) but, a different key should be used by the cryptosystem (requirement 11), so that a flaw in the steganographic algorithms will not compromise the security of the encryption key.

Video files are densely packed with data and ideally a steganographic solution should be capable of embedding information in as much of this data as possible. Therefore, it is highly desirable that more sophisticated embedding schemes are investigation to fully exploit the properties of video files (requirement 14, 15 and 16). The implementation of more complex methods is only desirable and not mandatory because of the unknown complexity of manipulating video data.

3.3.3 Steganalysis

The requirements for the steganalysis aspects of the software are initially basic. The proposed steganalysis requirements (8, 9 and 17) define tools that will be helpful to a user who wants to perform steganalysis – the process of steganalysis is not intended to be purely computerised. This approach to steganalysis has been reached based on the knowledge gained by conducting the literature survey. As with cryptography, there is no "special formula" or algorithm for detecting steganography, therefore generic tools capable of providing useful analysis (regardless of the embedding scheme) have been specified.

3.4 Testing Strategy

Several key components of the system will depend on other components and testing these will not be straightforward. For instance, to be able to test the encoder works properly we will either need the steganalysis tools to be developed so that this can be manually verified, or the decoder will have to be developed to extract the message. Once the encoder and decoder are developed any supported video file and message can be used to test these two components of the overall system.

Some components, such as the AES cryptosystem, will be utilised by other components of the system but will not depend on any itself. Unit testing can be used to test the cryptosystem, and parts of the encoder and decoder will be unit testable.

System testing will be needed to test all of the components working together. Video files produced by our system will have to be manually tested to determine that the files are still capable of playback, and that any degradation of quality is reasonable.

In the majority of situations standard testing approaches such as unit testing and system testing will enable us to test the functionality of the system. This is especially true for those features which can be quantitatively measured.

By definition, a steganographic scheme should make it difficult, if not impossible, for an adversary to determine the presence of a message. To an extent, the difficulty associated with performing steganalysis on a steganographic scheme should also be considered a testable parameter, however, this does heavily overlap with evaluation.

3.5 Evaluation Strategy

We will primarily evaluate our system by cross-referencing the final product with the requirements that are defined in this chapter. For a successful implementation our system should implement all of the mandatory features.

We can further evaluate our project by comparing it to third-party software projects. We have found only three software applications[1,2,3]. that are capable of performing some method of video steganography. We can compare our software product with these applications to evaluate the overall quality of the solution that we have produced.

Throughout the project we will evaluate the effectiveness of our steganographic schemes by using steganalysis. However, just because we cannot spot flaws and trends in our schemes does not mean that they are secure and free from error, therefore we will attempt

[1] *Our Secret* – `http://www.securekit.net/oursecret.htm`
[2] *OpenPuff* – `http://embeddedsw.net/OpenPuff_Steganography_Home.html`
[3] *MSU StegoVideo* – `http://www.compression.ru/video/stego_video/index_en.html`

to use third-party tools to perform further steganalysis. *StegSecret*[4] is an open-source tool capable of performing steganalysis on various media formats (including video) that can be used to evaluate the effectiveness of our steganographic schemes.

Whilst most of the evaluating of the steganographic techniques will have to be done by ourselves using our own steganalysis (together with third-party tools), we can also evaluate aspects of the software product based on feedback from other users. By releasing the application to test users we can ask them to evaluate the graphical user interface and ease of use of the application. Involving external users will help us to evaluate whether we have met some of the more subjective, user-oriented requirements such as requirements 22 and 23. Furthermore, we hope that by utilising feedback from end-users, we will be able to not only ensure that we have met all of the requirements defined in this chapter, but have also implemented a practical and user-friendly application.

3.6 Chapter Summary

In this chapter we have outlined the requirements of our system and our proposed testing and evaluation strategies, basing our analysis in part on knowledge gained in conducting the literature survey (see Section 2).

Detailed consideration has been given not only to the requirements of the steganographic scheme, but also the practical implications of attempting to develop a usable steganography application.

As a result of our analysis we have also identified that the amount of information relating to the implementation of video transcoding and video steganography in the public domain is extremely limited, and in the case of video steganography, almost non-existent.

In Section 4 we will discuss how the designs and implementations have been adapted following the requirements and analysis undertaken in this chapter.

Chapter 4: Design and Implementation

Whilst there are numerous software development models the nature of our project has simplified the selection process. Software development processes such as the waterfall and V model have been disregarded because of their rigid structures, opting for an agile software development approach which is more reactive and iterative in its nature.

Developing a steganographic technique that is capable of hiding data in a video file lies at the heart of this project, and it is desirable that the resultant solution is as secure as possible. Agile design methods allow us to evaluate and improve on our implemented techniques as we progress. Furthermore, our survey of the literature and our preliminary research (see A.2) yielded extremely limited information on how to design and implement a video steganography system of this nature. With this in mind, agile development would allow us to make substantial changes to the design and implementation at any point – something which cannot easily be achieved with other software engineering models.

[4] `http://stegsecret.sourceforge.net/`

It is at this juncture that we would like to point out that our research has been substantially more experimental than we could have possibly foreseen. Our choice of an agile development process has been a proven blessing on numerous occasions. While the majority of this chapter will document the design and implementation of our final solution, we will also highlight how our design and implementation have changed with the iterations. Arriving at the final state of implementation has been a complex process and several important lessons have been learnt along the way.

4.1 Design Decisions

In Appendix A.3 we discuss aspects of our initial research which involved a broad investigation of steganography (using audio, image and video containers). The insight provided by this research influenced the design decisions that follow.

In our preliminary research we identified, explored and implemented some of the methods used to embed messages in audio and image files. This approach allowed us to develop knowledge of how to manipulate spatial (image) and temporal (audio) data for the purposes of steganography. The resulting tools allowed us to embed messages in WAV audio files and PNG images.

The bulk of our preliminary research was dedicated to developing an understanding of video coding and manipulation. As we discuss in sections 4.1.1, 4.2.2 and A.3 this early research used Xuggler and Java. Although Xuggler and Java eventually proved unfit for our purposes, this allowed us to demonstrate how to manipulate video frames, and how subtle pixel colour manipulations were not possible because of the transform domain based compression that is used to represent video frames.

4.1.1 Libraries

Given the timescale for this project it would have been impossible to develop a video coding[1] tool with the level of functionality that we required for our project. Developing a video coding tool from scratch, that was capable of coding frames of a video stream would be time consuming and complex in its own right, given the coding process that is used by the majority of video codecs (Section 2.3). Therefore we opted to use FFmpeg[2] – a comprehensive audio/video library written in C for manipulating multimedia data.

As we explain in sections 4.2.2 and A.2 we also used Xuggler [3], a Java wrapper for the FFmpeg library. This was abandoned because it did not allow us the low-level manipulation that is required by our project. Java and Xuggler were later reintroduced to our project for the sole purpose of developing the GUI part of our solution (which is discussed in Section 4.2.3).

4.1.2 Codecs

As our system will utilise the FFmpeg library we decided to limit our choice of codecs to those that use macroblocks and are already supported by FFmpeg. In our literature survey

[1] The term "video coding" is used to refer to the process of transcribing a video into bytes to be stored on disk. This should not be confused with embedding data in a video.

[2] `http://www.ffmpeg.org/`

[3] `http://www.xuggle.com/xuggler`

we highlight how video affords a far superior steganographic capacity when compared to other media formats such as audio and images.

H.264 is the most popular video codec on the internet with at least 66% of all online videos being encoded with H.264 [Sch10]. BBC iPlayer, DaCast, PlayStationStore Movies & TV Shows, Vimeo, Vudu and YouTube all use H.264 for the encoding of HD video [Ros08, DaC, Dip08, Vim, Stu08]. In addition to the factors mentioned above, H.264 is also the most common video codec used by modern camcorders and cameras [Cas10].

4.2 System Design

4.2.1 Overview

The design of this system has been split into two major components:

- The steganographic tool
- The graphical user interface

By separating these two major areas of the system, we can develop them using the programming languages, libraries and tools that are best suited for the given task. This concept is inspired by unix-based operating systems, where graphical user interfaces are often used as a "middle man" between the user and a command line tool.

Furthermore this structure places the core steganographic tools in a single executable that could be re-used for other projects and by others. For instance, there are numerous video conversion programs available on the internet that use the FFmpeg binary for converting a video. In general, these tools simply provide a GUI that interacts with the standard input and output of the FFmpeg command-line binary.

4.2.2 Command Line Tool

The command line tool will be the main product of this solution and will contain the steganographic algorithms for embedding and extraction, as well as analysis tools for steganalysis.

Programming Language

As stated in requirement 21, our system needs to run as a cross-platform solution under Linux, Mac OS and Windows. Java would easily allow us to develop a cross-platform solution, as implementations of the Java Virtual Machine (JVM) exist for these platforms.

Unfortunately, our early research highlighted limitations with developing a video steganography system in Java using Xuggler (Section A.3.3). In short, Xuggler did not provide the low-level functionality needed to manipulate a video file. With this in mind we considered other Java libraries (including Java Media Framework (JMF)), and even the possibility of writing our own wrapper using the Java Native Interface (JNI)[4]. Whilst using the JNI would be an option, it was still unclear to us at the time whether the limitations we encountered were solely due to the development state of Xuggler, or whether FFmpeg did not support what we wanted it to do.

[4] `http://docs.oracle.com/javase/6/docs/technotes/guides/jni/`

Ultimately, our choice of programming language was governed by the fact that our project relies heavily on video manipulation. We therefore opted to develop our video steganography tool in C, despite having no prior knowledge of C. C would allow us to do what we could not do with Java because of the FFmpeg library and the low-level functionality afforded by interfacing directly with the library. Developing a system in a new programming language was certainly risky, however, as this would introduce its own learning curve. Whilst C is widely supported on Linux, Mac OS and Windows, developing a cross-platform solution in C is more complex than it is in Java, as you have to consider the differences in the operating systems.

With Java, bytecode compiled on one processor architecture will run on any other processor because the JVM provides an operating system independent platform on which the bytecode can run. The complexity of dealing with different processor architectures is reserved for the given JVM implementation.

On the other hand, C is compiled for a specific processor architecture, and the programmer must consider the differences between operating systems and processor architectures. Whilst this makes the process of developing a cross-platform solution more complex, it is still achievable.

It should also be noted that C will provide far superior performance than any solution produced using Java, because Java solutions have to run on top of the JVM. Naturally, the JVM will introduce an overhead as this runs on the processor and acts as a "middle man". In addition to this, the performance of a solution developed by us will be limited by the performance and optimisation of the JVM implementation. In contract, C is a highly efficient programming language that is executed directly via the processor, and as such it is used heavily by other computationally intensive domains such as 3D graphics.

Commands Overview

The command line tool will accept any of the commands listed below. Angle brackets indicate required arguments and square brackets are used to denote optional arguments.

- `-cryptographic-algorithms`
  Lists the names of the supported cryptographic algorithms.

- `-decode <inputFile> <outputFile> <scheme> [key]`
  Decodes the embedded payload of the `inputFile` to the `outputFile` using the specified steganographic `scheme` and `key`. `scheme` should specify one of the scheme names returned by `-schemes`.

- `-decrypt-test-vector <algorithm> <blockMode> <keySize> <key> <cipherText> [initalisationVector]`
  Runs the decryption process for the given algorithm with the specified parameters to verify the correctness of the decryption algorithm. `[initalisationVector]` is an optional parameter that is only needed when the CBC `blockMode` is used. `algorithm` should specify one of the scheme names returned by `-cryptographic-algorithms`.

- `-encode <inputFile> <objectFile> <outputFile> <scheme> [password]`
  Encodes an `objectFile` into a container object (`inputFile`) using the specified steganographic `scheme` and `password`. `scheme` should specify one of the scheme names returned by `-schemes`.

- `-encrypt-test-vector <algorithm> <blockMode> <keySize> <key> <plaintext> [initalisationVector]`
  Runs the encryption process for the given algorithm with the specified parameters to verify the correctness of the encryption algorithm. `[initalisationVector]` is an optional parameter that is only needed when the CBC `blockMode` is used. `algorithm` should specify one of the scheme names returned by `-cryptographic-algorithms`.

- `-meta-data <inputFile>`
  Provides meta data information for the specified `inputFile`.

- `-mv-dump <inputFile> <frameNo>`
  Outputs the motion vectors for each macroblock of the specified `frameNo` of the `inputFile`.

- `-schemes`
  Lists the names and descriptions of the steganographic schemes.

- `-transcode <inputFile> <outputFile`
  Takes in a video `inputFile` and uses the transcode mechanism (see Section 4.2.2) to produce a facsimile `outputFile`.

- `-unit-test`
  Runs all unit tests

- `-vs-overview <inputFile>`
  Iterates the video stream of the specified file and outputs the type of each frame.

- `-h <command>`
  Provides help on how to use the program. If a specific `<command>` is specified, help on how to use that command will be outputted.

The command line tool will work using the standard input and standard output. All input will be provided as command line arguments. The tool will use the return code (exit status) to denote whether the operation succeeded or failed. We will adopt the widely used convention of returning zero upon success and a non-zero value (by default 1) upon failure.

Transcoder

The process of transcoding a video file will require demultiplexing the original input file to distinguish the audio data from the video data. The separate audio and video data can then be processed and encoded (multiplexed) back into the output file.

To be able to understand how the transcoding process works, it is important to define the meaning of the terms *packet* and *frame*. Typically a packet is a formatted unit of data; in the context of video coding this still holds, however, a packet is the compressed data that is taken directly from the input. A frame contains the decoded (uncompressed/raw) audio or video data. In the case of audio the frame contains audio samples and for video the frame will describe a picture. It is also important to note that depending upon the codec used and the frame type, it is possible for a frame to span multiple packets of data, and as such a frame cannot be decoded until all of the necessary packets have been processed.

Similarly, it is important to check the cache for frames once all packets have been read from the input file. Although a frame can span multiple packets, it is also possible for a packet to contain multiple frames. Without checking the cached frames at the end of a video file these frames can be ignored.

Figures 4.1 illustrates the design of the transcode process and figure 4.2 shows how packets from the input video are decoded and managed. The general transcoding process shown in figure 4.1 is relatively straightforward. The decoding process (figure 4.2) is more complex.

A Presentation Time Stamp (PTS) is used to synchronise the different streams of a video file. Correctly setting the PTS is imperative as failing to do so will result in audio and video streams that are out of synchronisation. Incorrect PTSs can also cause an inflation of the video file size. The PTS of a video uses a 90 Khz clock relative to the start of the video. During the decoding process the audio and video streams for the video file are decoded according to a common time base. The common time base generally depends on the decoder clock of one of the streams [Tek00].

Encoding and Decoding

The encoding and decoding of data will be performed by modifying the motion vectors of a given video frame by means of a callback. This approach is straightforward in the sense that the transcoding mechanism detailed in figures 4.1 and 4.2 can be used to either transcode a video normally or embed the contents of an object file into the container.

We have opted for this very abstract approach as it will allow us to develop the different aspects of the system independently, furthermore this approach will allow us to easily add different steganographic schemes allowing the user to choose which scheme they would like to use. Our callback will be implemented directly into the FFmpeg library.

Whilst the `AVFrame`[5] struct does provide a `motion_val` attribute we will only be able to use this for the decoding of the message as `motion_val` is set by the `avcodec` component of FFmpeg. Therefore, `avcodec` will need to be adapted to handle a callback to our code.

The motion vector to modify and the frame number will be passed to our callback. Out callback will then use this information to appropriately modify the motion vector that is passed to it.

Figures 4.3 and 4.4 illustrates how our system can easily be expanded to include different steganographic schemes. The `Encoder` is used to manage the object file (method to be embedded) and the chosen steganographic scheme.

`FirstMbXEncoder` was the first steganographic scheme that was implemented by our system. This scheme works by manipulating the X-component of the first motion vector of each frame. Following this a Y-component variant of this scheme was also developed (`FirstMbYEncoder`).

`stegEncodeMv` is our callback function that is called from inside the `avcodec` library, this function will chain a callback to the relevant encoder modules, which is determined by `getStegEncoderMode`. The two steganographic schemes that are implemented in the system thus far implement identical functions and attributes, however they differ significantly in the implementation of their chainable callbacks (`firstMbXEncodeMv` and `firstMbYEncodeMv`).

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

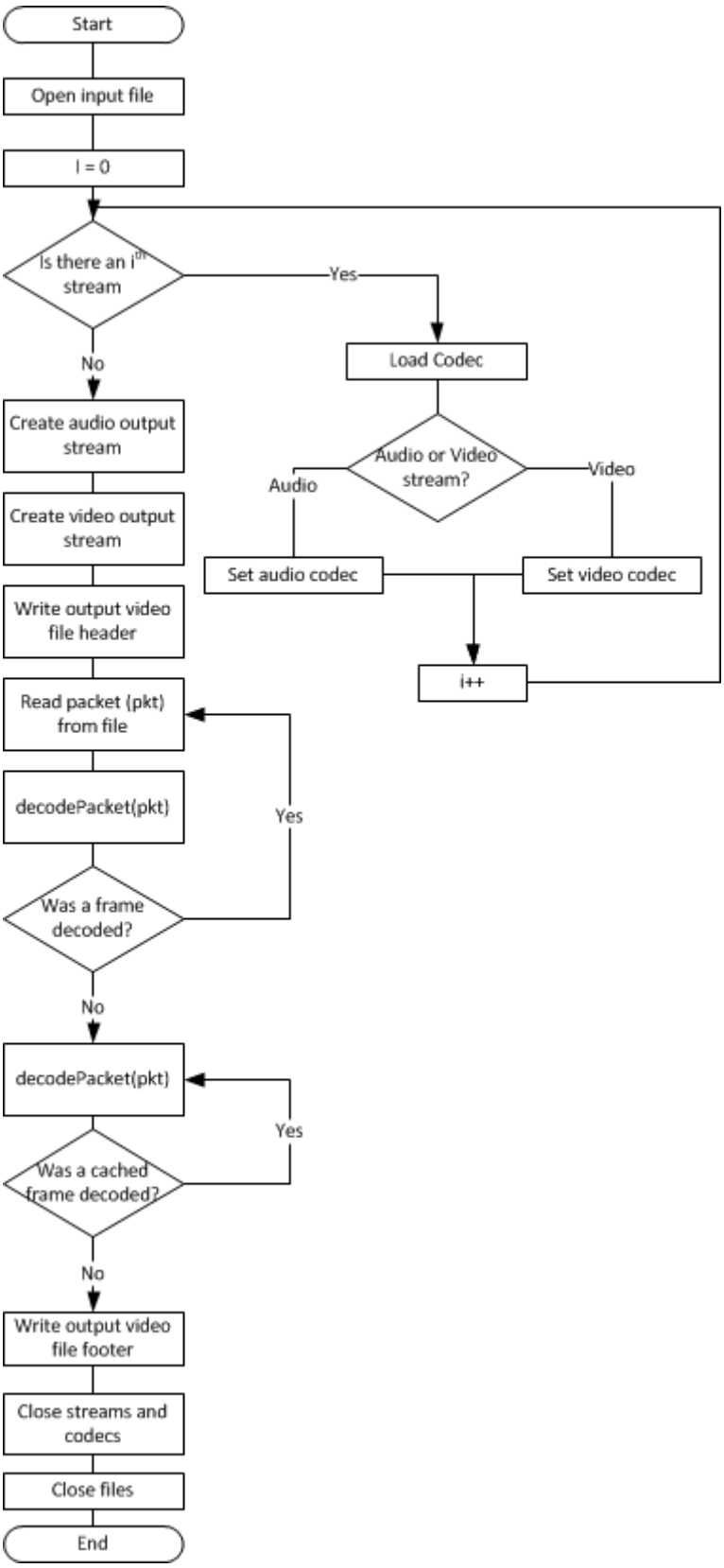


Figure 4.1: Transcode process flow chart

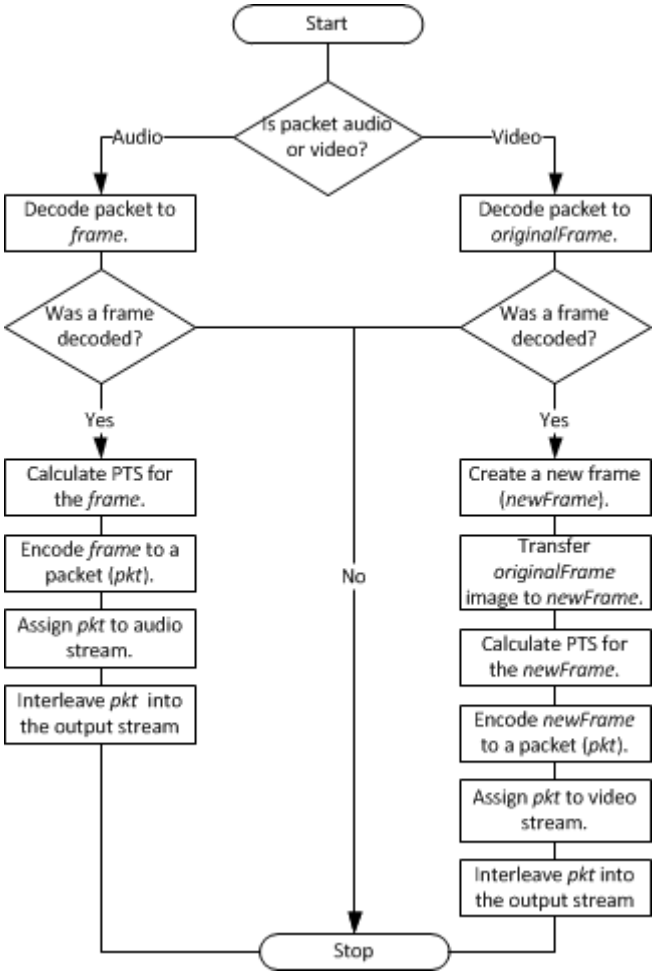


Figure 4.2: Transcode process: `decodePacket` flow chart

Figure 4.3: Encoder architecture overview

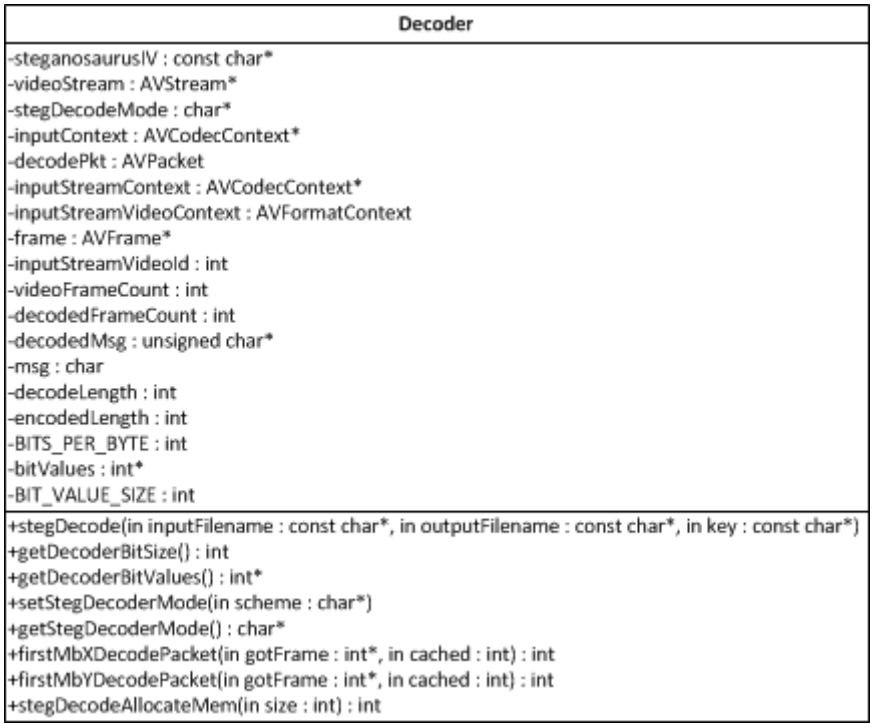


Figure 4.4: Decoder architecture overview

Developing different steganographic encoding schemes is straightforward, as long as the `Encoder` is aware of any additional coder modules. The decoding process is even simpler as this is contained within a single `Decoder` (figure 4.4). The decoder works by iterating over the packets of the specified video file and calling the relevant packet decoding method.

Steganalysis

As stated in Section 3.3.3, our solution will provide tools to aid the user in performing manual steganalysis. Requirements 8, 9 and 17 all denote steganalysis features that should be implemented in our application.

Requirements 9 and 17 will be achievable manually using the `-mv-dump` feature of the command-line tool, therefore the GUI will simply use this command to allow the user to easily compare neighbouring frames and frames from different videos.

There is no command or implementation for performing the functionality of requirement 8 in the command line tool, and this is a deliberate design choice. We have decided to incorporate the playback and viewing of video frames in the GUI only, this avoids the issue of trying to handle graphical tasks in the command line tool and contains all graphical functions to the GUI.

AES Cryptosystem

The design of the AES cryptosystem module will be governed extensively by its formal specification. Our design intention is to implement an AES Cryptosystem that is fully compliant with the FIPS-197 standard [Nat01]. In Appendix B we discuss at length the design and workings of FIPS-197 compliant AES.

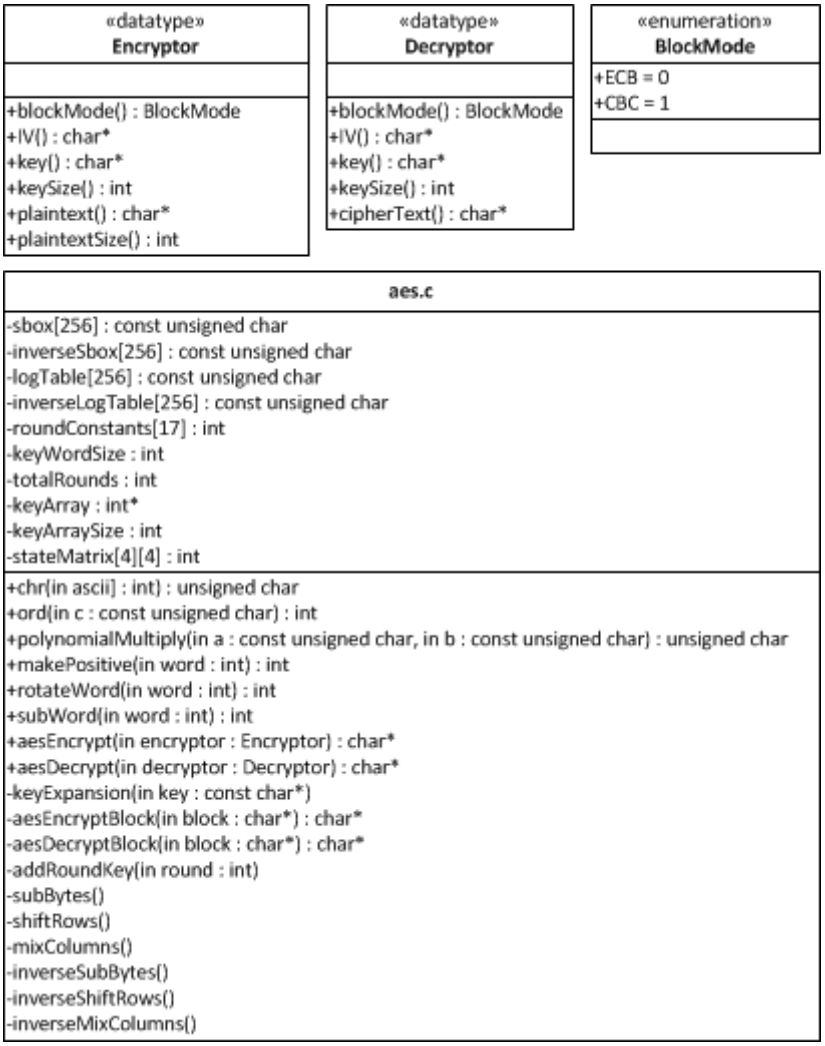


Figure 4.5: AES cryptosystem overview

In addition to AES we also discuss the modes of operation that can be used with symmetric block ciphers such as AES. Our AES crytosystem will use the Cipher Block Chaining (CBC) block mode (see Section B.3.2) to encrypt data. This has been chosen over other methods, such as Electronic Code Book (ECB), because CBC adds a level of randomisation that prevents patterns and trends exhibited in plaintext persisting in the cipher text.

Block ciphers, by definition, encrypt data in blocks (or groups) of bits and as such, our system will encrypt the steganographic message to be hidden into memory before starting the embedding process. Similarly, when a message is extracted from a steganographic container, the cipher text will be completely extracted before commencing decryption.

Figure 4.5 shows the UML design for our cryptosystem. The vast majority of the operations in our cryptosystem design correspond to AES operations discussed in Appendix B and the FIPS-197 specification as this implementation has been designed to be fully compliant.

4.2.3 Graphical User Interface

Programming Language

Having the GUI as a separate application comes with the advantage that it does not have to be developed in the same programming language. We initially investigated using C for the GUI (as well as the command line tool), however this proved to be impractical. Producing a GUI in C that was capable of media playback would have relied on using several third-party libraries (such as GTK+[6] and SDL[7]). GTK+ is the only library that provided all of the GUI components that our GUI needed and was compatible with Linux, Mac OS and Windows. Unfortunately, after some further investigation and experimentation with these libraries we found that the GTK+ library had very poor Mac OS integration. GTK+ was unable to interact with the Mac OS menu bar which caused serious problems as the original designs relied on a menu bar to access the tools of the application.

Any good GUI should be multithreaded so that the GUI can be updated as operations complete. With C there is no uniform cross-platform API for managing threads. Although Unix based operating systems (such as Mac OS and Linux) use *pthreads*[8], Windows uses an entirely different threading API. Circumventing this issue would require us to use a third-party library or develop our own mechanism for supporting multithreading cross-platform. Given the native integration issues with GTK+ and the problem of multithreading we deemed that C would be an inappropriate programming language for completing a GUI capable of fulfilling our designs and requirements.

For the purpose of developing a GUI we considered using Java and Xuggler. Given the results of our preliminary research (Section A.3) we had discarded these for the purpose of video manipulation. However Xuggler and Java are sufficiently suited for GUI development purposes. Java provides extensive GUI components such as the Swing[9] and AWT[10] packages which provide a far superior range of GUI tools than GTK+. Furthermore, Xuggler could be used for managing the playback of media at the GUI level.

Design

The final implementation of the GUI was to be developed in Java, utilises the functionality of the command line tool. Communication between the GUI and the command line tool is governed by the `Process`[11] and `ProcessBuilder`[12] classes. These classes allow a Java application to execute external processes (using command line arguments), whilst redirecting and managing the standard output and standard error streams.

Figures 4.6–4.14 and 4.16–4.18 show mockup designs for the interface. These mockups illustrate the general structure and layout of each window and dialog. Figures 4.6–4.8 show the main interface. From this window the user can encode messages into a video container, and decode a message hidden in a steganographic video. All other windows are modal dialog tools that are accessible from the main interface, gemerally via the menu (figure 4.8).

[6] GTK+ (GIMP Toolkit) – `http://www.gtk.org/`
[7] SDL (Simple DirectMedia Layer) – `http://www.libsdl.org/`
[8] `https://computing.llnl.gov/tutorials/pthreads/`
[9] `http://docs.oracle.com/javase/7/docs/technotes/guides/swing/`
[10] `http://docs.oracle.com/javase/7/docs/technotes/guides/awt/`
[11] `http://docs.oracle.com/javase/6/docs/api/java/lang/Process.html`
[12] `http://docs.oracle.com/javase/6/docs/api/java/lang/ProcessBuilder.html`

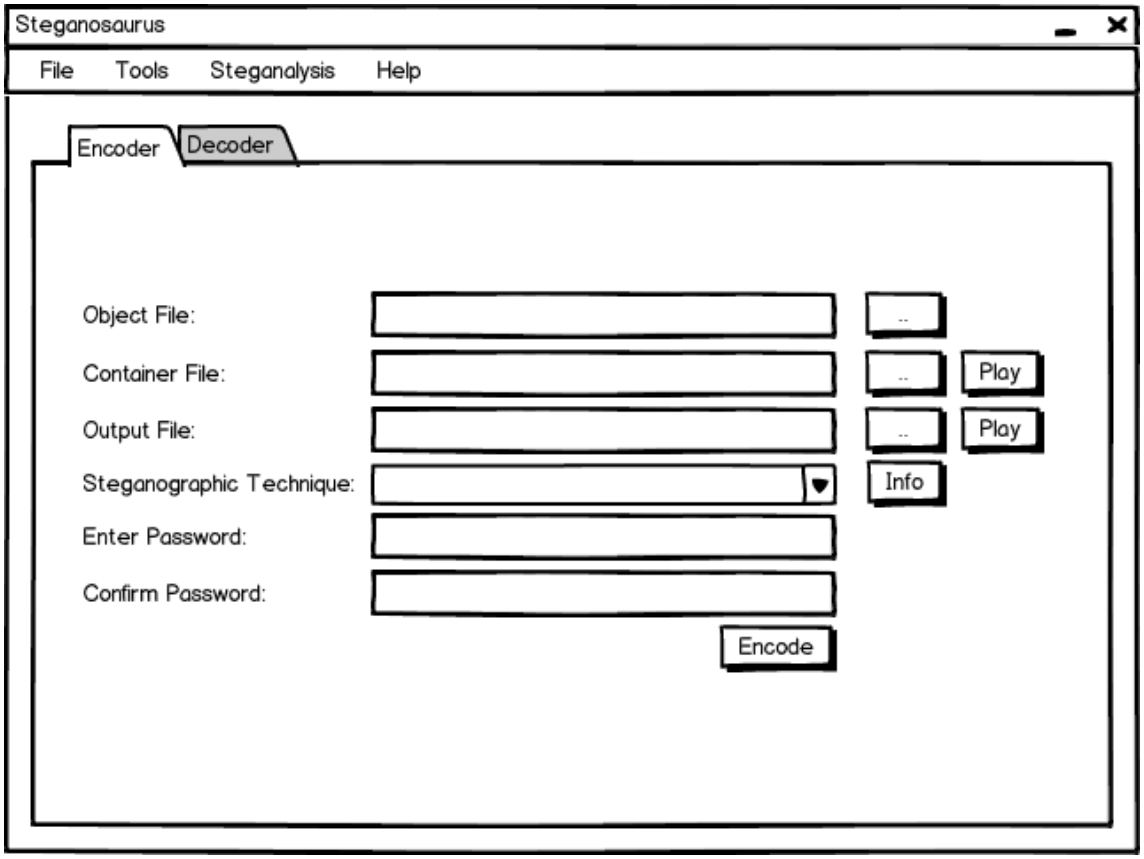


Figure 4.6: GUI – Main Interface (Encode)

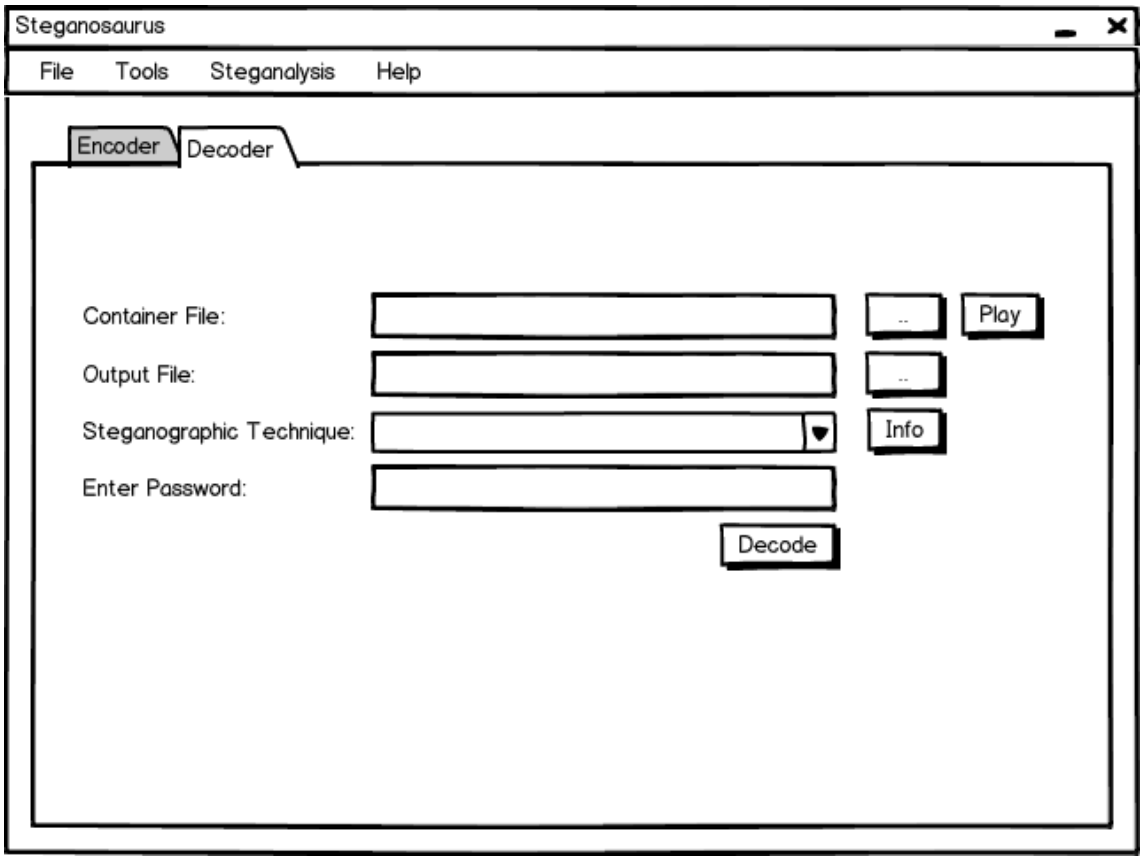


Figure 4.7: GUI – Main Interface (Decode)

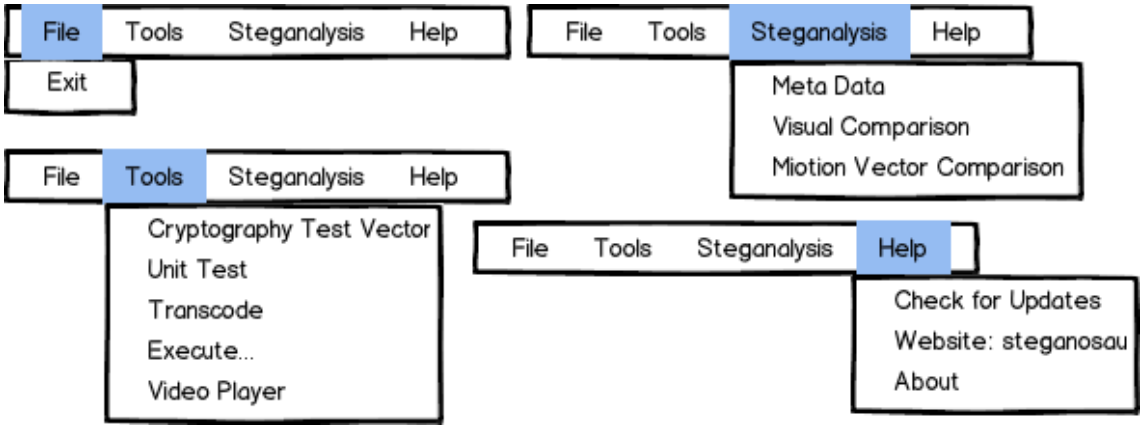


Figure 4.8: GUI – Main Interface – Menu Structure

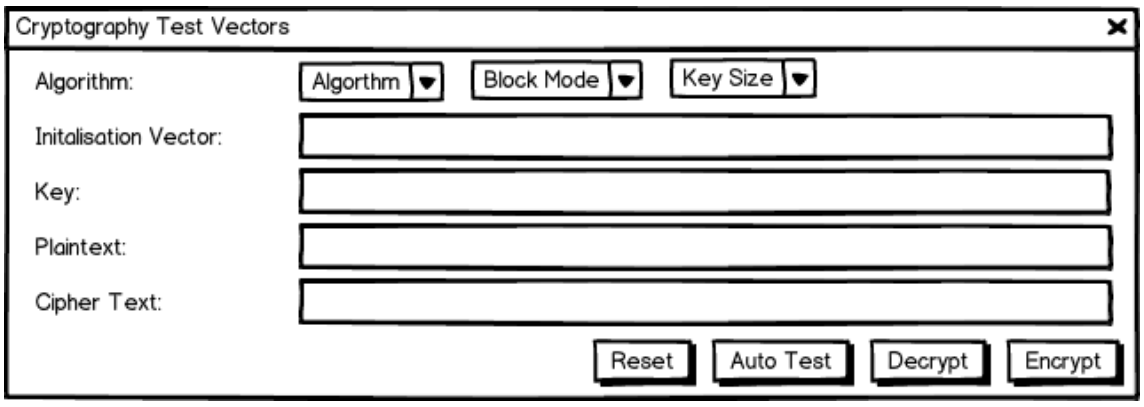


Figure 4.9: GUI – Cryptography Test Vectors Dialog

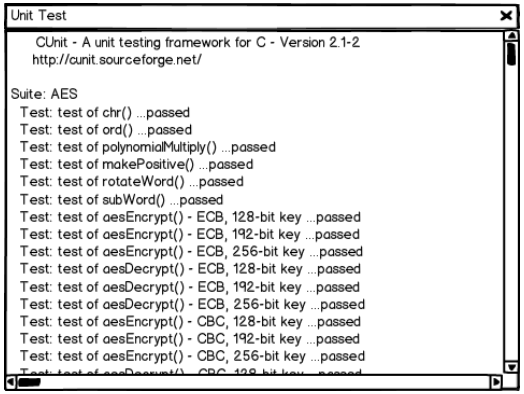


Figure 4.10: GUI – Unit Test Dialog

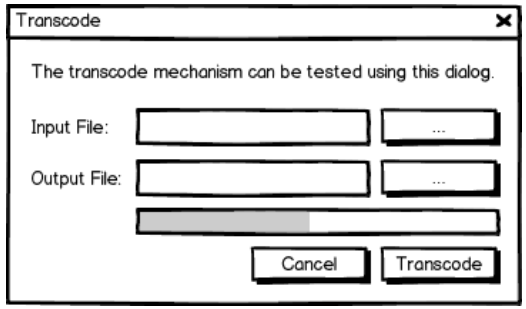


Figure 4.11: GUI – Transcode Dialog

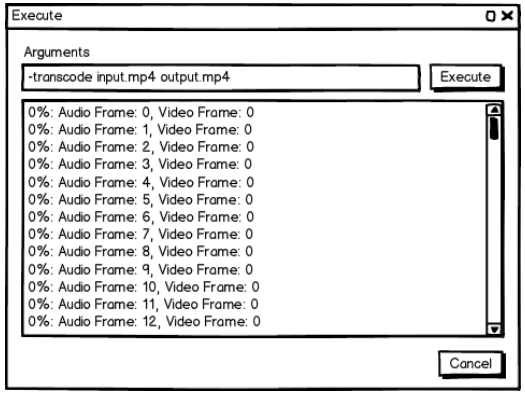


Figure 4.12: GUI – Execute Dialog

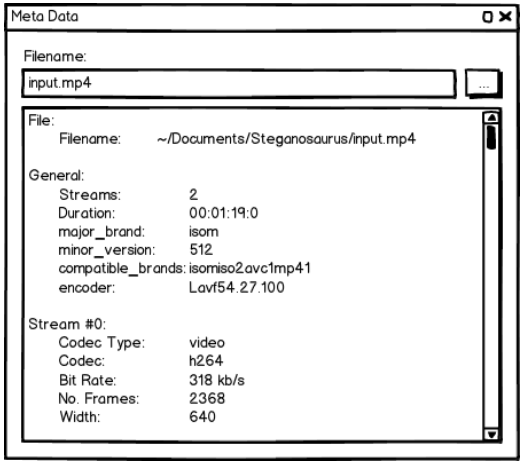


Figure 4.13: GUI – Meta Data Dialog

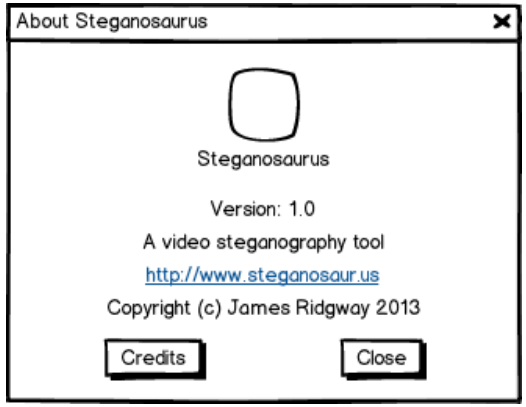


Figure 4.14: GUI – About Dialog

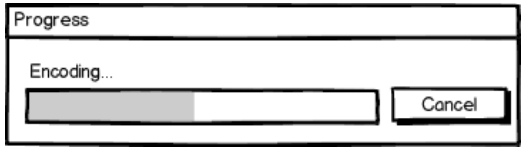


Figure 4.15: GUI – Progress Dialog

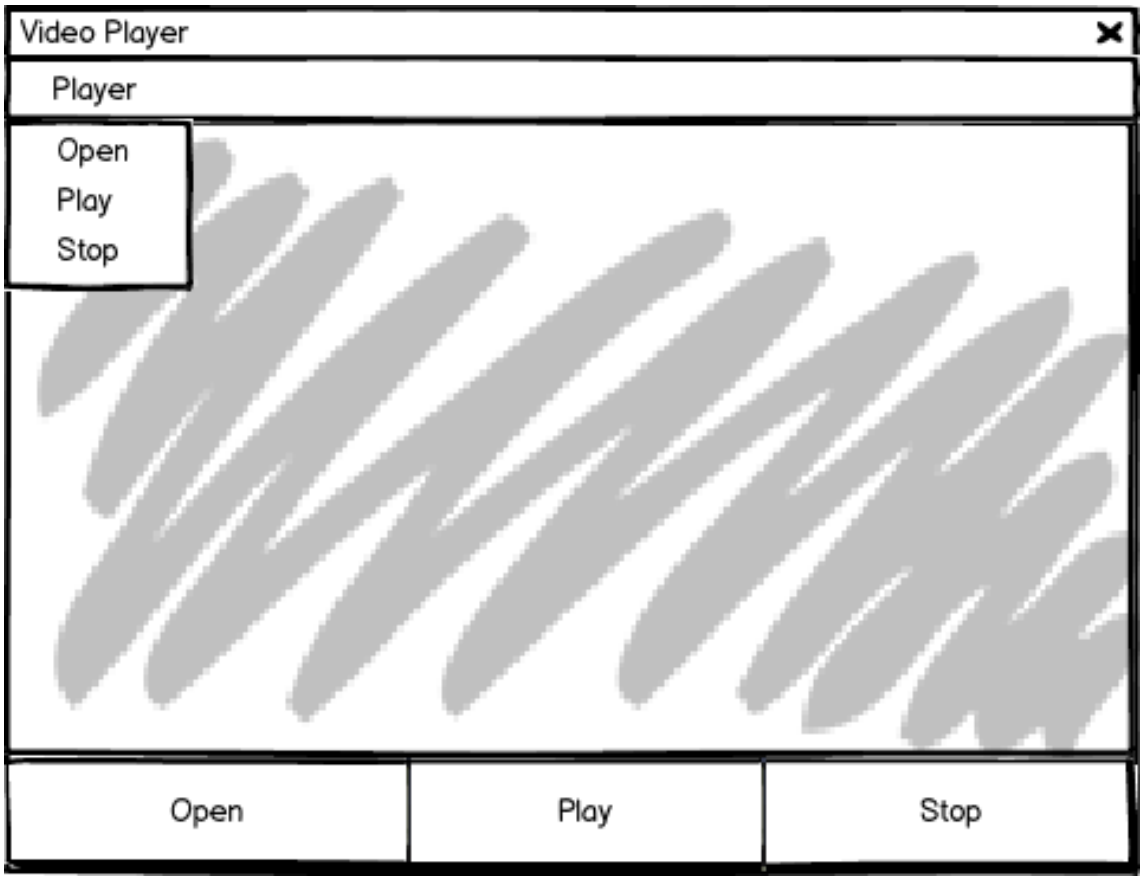


Figure 4.16: GUI – Video Player Dialog

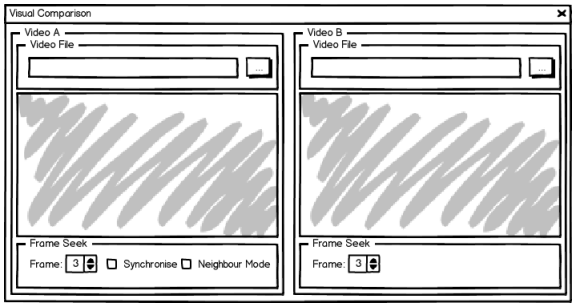


Figure 4.17: GUI – Visual Comparison Dialog

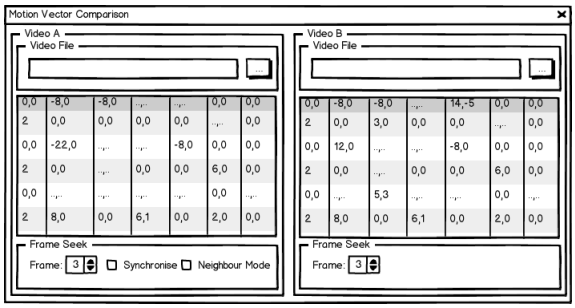


Figure 4.18: GUI – Motion Vector Comparison

4.3 Implementation

4.3.1 Overview

The implementation of this system was not straightforward, and numerous setbacks meant that various components of our system were re-written and re-designed to overcome these unforeseen hurdles.

The bulk of our system was implemented in C as a command line tool. A Java GUI was also built to interface between the end user and our command line application. Implementing our system so that it worked across Linux, Mac OS and Windows (requirement 21) was not as easy as we first thought, and as a result subtle adjustments had to be made to our implementation to ensure that it would run on the required operating systems.

The final implementation of our solution used several libraries. The main command line tool incorporates a modified version of the FFmpeg library, and in addition also links with CUnit – a unit testing framework. Our Java GUI is implemented using Swing and includes Xuggler.

4.3.2 Command Line Tool

The command line tool we implemented includes a modified version of the FFmpeg library, and as such is written entirely in C. The command line tool contains five core components:

- Transcode Mechanism
- Steganographic Encoder
- Steganographic Decoder
- AES Cryptosystem
- Steganalysis Components

Transcode Mechanism

This component provides an underlying mechanism upon which the Steganographic Encoder is built. The transcode mechanism takes an input video file, parses the packet data, and outputs a new video file using the decoded packets.

The transcoder is built using functions of the `avcodec` and `avformat` libraries of FFmpeg. The specified input file is first scanned for audio and video streams by iterating over all of the media streams present in a file. When appropriate audio and video streams are found their respective codecs are loaded into memory using the `avcodec_open2` function. Loading a codec into memory creates an `AVCodecContext` that defines context specific information about the codec (attributes such as: bit rate, sample rate, pixel format). This context specific information is used to confirm the audio and video streams for the output video file which are represented by an `AVFormatContext`. Once the streams have been configured the header (which details the stream setup) has to be written to the output file using `avformat_write_header`. With a prepared output file it is now possible to iterate over the packets of data in the input file. A packet can contain multiple frames, so it is important to check that no more frames can be decoded from the last packet once the file

has been exhausted. Once all of the packets have been decoded the footer can be written to the output file using `av_write_trailer`.

Decoding packets from the input file for the purpose of encoding in the output file had its pitfalls. The frame that is produced by the decoder methods (`avcodec_decode_video2` and `avcodec_decode_audio4`) cannot be parsed directly to the encoder without a certain amount of preparation. Decoded frames do not contain PTSs as these specifically relate to the video context. Before a decoded frame can be re-encoded it needs, at the very least, each frame had to have a correct PTS, in addition to this the video frames contained meta data that interfered with the encoding of the video.

Setting an accurate PTS and Decompressed Time Stamp (DTS) was crucial, and failure to do so resulted in either no video image, or unsynchronised audio and video streams. The resultant solution involved scaling a counter using `av_rescale_q`.

For some as yet unidentified reason, frames decoded by `avcodec_decode_video2` could not be parsed directly to the encoder, because they contained meta data that interfered with the encoding. The meta data caused the image to be pixelated and the `avcodec` library to throw various warnings. This problem was eventually solved by copying the raw image data to a new `AVFrame` struct.

Steganographic Encoder

Our command line tool was originally developed so that it linked to the FFmpeg component libraries (`avcodec`, `avformat`, etc.) for the transcoding process. This principle worked well for ordinary transcoding, therefore, when we started to develop the encoding process we looked at modifying the `AVFrame` that was parsed to the `avcodec_encode_video2` method for coding into a compressed packet. From the FFmpeg documentation it appeared that we needed to modify the `motion_val` attribute of the `AVFrame` before calling the encode method. Despite our endeavours the alterations we were making to `motion_val` were not being reflected in the coded packet. It eventually became apparent that we would be unable to manipulate the motion vectors by changing the `motion_val` attribute prior to the `avcodec_encode_video2` method.

After some further investigation and unsuccessful attempts to modify the compressed packet, we concluded that the only solution would be to modify the FFmpeg source code. Our efforts then shifted to dissecting the 850,000+ lines of code that comprise the FFmpeg codebase. Eventually we deduced that by adjusting the behaviour of `ff_estimate_p_frame_motion` in `libavcodec/motion_est.c` we could manipulate the motion vectors of macroblocks in P-Frames.

Now that we were able to modify the motion vectors of a frame we set about developing a callback method for the steganographic encoder. This would require us not only to modify the FFmpeg source code, but we would also have to include the source code as part of our project. We therefore merged our work thus far with the FFmpeg source code.

At this point in time (April 2013), our steganographic encoder only modifies the X component of motion vectors by means of a bit mask. A callback is made to the `stegEncodeMx` of our main `encoder.c` file, which is responsible for chaining callbacks to appropriate methods depending on the chosen steganographic scheme. The `stegEncodeMx` method takes a `MpegEncContext` and the integer value of the X component of the motion vector. From the `MpegEncContext` we are able to derive attributes such as the frame number that will allow us to embed the relevant bits.

The approach mentioned above was not without its flaws, our tests and experiments showed that only small messages of no more than 30 characters could be embedded, and on average 50% of the bits that were embedded would flip causing an inaccurate decoding of the hidden message. Further experimentation showed that different bit masks provided little improvement and in some instances substantially decreased the number of recoverable bits.

Eventually, we discovered the following issues, in order, that contributed to the erroneous embedding of data. Implementing our callback in the `ff_estimate_p_frame_motion` method of `libavcodec/motion_est.c` caused the covert data to be embedded before the quantisation process (Section 2.3.2). As a result the range of motion vector values was being further manipulated and altered, in some instances obliterating our changes. After further exploring the FFmpeg source code we moved our callback from the `ff_estimate_p_frame_motion` method of `libavcodec/motion_est.c` to the `encode_mb_internal` method of `libavcodec/mpegvideo_enc.c`, as a result our motion vector manipulation occurs just before the coding process.

After another round of analysis we realised that the values returned by our bit mask in the decoder varied without any noticeable pattern. For instance if our embedding scheme uses a bit mask of 7 to encode a "1" bit the we would expect a bit mask of 7 to return 7 or 0 depending on whether a "1" or "0" bit is encoded. We found that we would also get a "4" or "3" randomly returned, and as these were no-zero values they were interpreted as "encode a 1 bit". This bit mask issue was attributed to the coding process of a motion vector in which a right-shift operation is performed. This issue was thus combatted by using a different decoding bit mask, we switched to using an encoding bit mask of 2 would ensure that a decoding bit mask of 1 would extract the same bit value.

The refinements that we had applied to our original encoder architecture worked almost all the time. The final bit flipping issue was attributed to macroblocks that were marked as having no motion vector. It appeared that were able to modify the motion vectors of macroblocks that had been flagged for no motion vector coding. As a result, we simply skipped any macroblocks marked as having no motion vector. By moving the encoder callback, compensating for bit shifts and avoiding motion-vector-less frames our transcoding mechanism was capable of embedding data in video. Before the transcoding and embedding process begins the object file (containing the message to be embedded) is encrypted and stored in the encoder. As the video is transcoded bits are read from the encrypted buffer and are embedded according to the steganographic scheme.

Steganographic Decoder

The steganographic decoder is to an extent far simpler as the logic is contained with in a single file. The decoder operates by decoding the video stream of a file and parsing the relevant packets to a packet decoding function, the choice of packet decoding function depends on the type of steganographic technique chosen.

The decoder only decodes data rom the video stream because the currently implemented steganographic techniques only hide data in the video stream. Implementing a decoder is quite a straightforward process as this only requires the implementation of a single packet decoding function.

The decoding of a video packet is performed by parsing `AVPacket`s to `avcodec_decode_video2` until a complete `AVFrame` has been returned. The AVFrame can then be processed by looking at the `pict_type` and `motion_val` attributes.

AES Cryptosystem

The AES cryptosystem component of our system was implemented as a generic module capable of taking an input in the form of a string and encrypting or decrypting to an output string. The cryptosystem was successfully implemented after only two iterations – unit tests highlighted a minor bug in the implementation of the `keyExpansion` method. The final implementation of the cryptosystem component conformed rigidly to the design illustrated in figure 4.5 and the exact FIPS-197 specification.

The AES cryptosystem is utilised by both the steganographic encoder and decoder. The steganographic encoder encrypts the entire object file (message to be embedded), before starting the transcoding and embedding process. The system behaves in this manner, rather than as a "stream" that encrypts portions at a time, so that the steganographic methods are not restricted to embedding in a linear fashion. This design choice was made so that future embedding schemes would be able to randomly access the encrypted data. Similarly, during the decoding process all of the encrypted data is extracted to memory first, before being decrypted and stored on the filesystem.

Although the steganographic system that we have designed was not been built with the purpose of streaming in mind, there are no known limitations of our design that would prevent this system from being used with streaming video.

Some modifications would need to be made to our system in order for it to work in a streaming context. A marker would need to be placed at the start and end of the embedded data to indicate where in the stream the embedded data would start and end. This would allow our system to extract information which has been embedded at some location in the video stream.

A simple implementation of this system would use a plaintext marker to indicate the start and end of the embedded message. The message, which could be encrypted, would then reside in the video between the start and end markers. This method is limited because it would allow an adversary to monitor a video stream for the plaintext markers. By encrypting the markers this would prevent an adversary from being able to scan the video file for these markers.

Steganalysis Components

Under Chapter 3 we defined two steganalysis tools as mandatory functional requirements:

- Requirement 8: 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.
- Requirement 9: for a specified frame of the original and steganographic video file, allow the corresponding motion vectors to be analysed and compared.

Requirement 8 is purely graphical in nature – allowing a specific frame from an original and steganographic video to be displayed side-by-side. Given the graphical aspect of this function it will be implemented in the GUI only – in Section 4.3.3 we discuss in some detail why we have made this implementation decision. This functionality is implemented using Xuggler as the library will allow us to extract video frames as an `Image` via the `IVideoPicture`[13] class. Frames from different video sources can be viewed side-by-side,

[13] `http://www.xuggle.com/public/documentation/java/api/com/xuggle/xuggler/IVideoPicture.html`

or, by altering the options of this feature it is possible to view two adjacent frames from the same video source side-by-side.

Requirement 9 is not graphical in nature, therefore this has been implemented in the command line tool and is accessible via the `-mv-dump` option. Parts of the decode logic from the transcode tool have been recycled for this task. As with the transcode mechanism a video stream is first located within the video file, once this is done the packets of the video stream are decoded to `AVFrame`s and two-dimensional motion vectors are calculated based on the motion values coded in the `motion_val` attribute. This feature then outputs a tabulated matrix of motion vectors (one per macroblock) along with the frame number and type.

In addition to the steganalysis features explicitly outlined in the requirement, another steganalysis function has been implemented that provides an overview of video frames in a stream by iterating over a video stream and outputting the frame number and type for every frame. This is a very simple feature which involves outputting a frame counter and the `key_frame` and `pict_type` attributes of `AVFrame` for each frame. This steganalysis tool was developed primarily as an analytical tool to aid us in resolving the challenges that we encountered in implementing the steganographic encoder, however, as this has proven to be significantly useful we have included this as an additional steganalysis tool. This feature is accessible via the `-vs-overview` option of the command line tool. An extra dialog has also been added to the GUI to allow the user to access the video stream overview from within the GUI application.

4.3.3 Graphical User Interface

Before implementing the GUI in Java (using Swing and Xuggler), we started to develop a GUI in C using GTK+ and SDL as an all-in-one application (combining the GUI with the steganography core). Unfortunately, the range of GUI libraries for C is very limited, however, GTK+ was chosen as the GUI library because it was the only library that offered the range of GUI components that our designs needed, and was compatible on all of the operating systems specified in our requirements.

For displaying video frames our intention was to use the SDL library. Although SDL is cross-platform compatible, we were unable to get our code to compile and execute properly on Windows and Mac OS. Not only was SDL problematic, but GTK+ proved to have poor integration for Mac OS. In our designs the menu bar is crucial for accessing the different tools from the GUI. GTK+'s native implementation with the Mac OS menu bar was buggy and did not work. We considered moving away from using menu bars in our application but all of the alternative designs we produced were less intuitive than the ones with menu bars. Instead of adapting the design of our GUI, we opted to change the language and libraries that we would use to implement it, we opted instead to use Java with the Swing package and the Xuggler library. Swing and Xuggler allowed us to produce a cross platform GUI without the problems we experience when trying to do this in C with GTK+ and SDL.

Furthermore, Java solved the problem of multi-threading which had been overlooked in the original designs. In C, threading is not cross platform compatible. Windows has its own threading API and Unix based systems (Linux and Mac OS) use POSIX threads (more commonly known as pthreads). If we had continued using C we would have had to either use a third-party library to manage threading, or implement our own cross-platform threading API. By the point that multithreading needed to be addressed we had already

switched the implementation of our GUI to Java. Java provides its own `Thead`[14] class that allows for intuitive and platform independent multithreading.

4.3.4 Cross Platform Compatibilities

Compatibility

Achieving cross platform compatibility for our command line tool was problematic, as the different implementations of the `gcc` compiler vary on how strict they are. As a result, this meant that our solution would often compile under Linux and Mac, but not under Windows. Linking to other installed C libraries such as CUnit is a prime example of how our Makefile would allow us to compile our code under Linux, but not any other operating system. After some experimentation we found that linking libraries cross platform was best achieved using `pkg-config`[15]. This tool provides a unified method for linking libraries regardless of the operating system. `pkg-config` will detect and return the appropriate library paths, include-paths and compiler flags for a given library.

In addition to the compile-time errors mentioned above, we experienced a substantial setback with the transcode mechanism when we realised that the solution we developed would only work on Linux – on Mac OS or Windows the transcoded video would contain horrible distortion in the audio stream. A substantial amount of time was invested in resolving this problem before a resolution was found. The sample format of the audio stream was not being set correctly, and as a result the number of sample bits used in the transcoded video was varying between operating systems.

Deployment

Significant consideration has also been given to how our solution would be deployed and installed across the various platforms. Our solution is provided for download via our projects website[16]. Naturally, different versions of the software exist depending on which operating system you want to use the system on. C code has to be compiled differently depending on the operating system and processor architecture. Therefore the key difference between the different software versions is the binary executable of our command line tool. The Java GUI we have implemented in compiled into a JAR, and this JAR is shipped with the software irrespective of the operating system.

Windows

The Windows version of our software is downloadable as an executable installer that has been created with Inno Setup[17] – an open source installer for Windows programs. In addition to this, our Java GUI is also wrapped in a Windows executable, so that our software is as easy and intuitive to use as possible. Most Windows software is provided as an executable and we felt that requiring the user to run a Jar to use our software added an unnecessary level of complexity that could be negated by wrapping the Jar in an executable. Furthermore, the executable wrapper can perform sanity checks and ensure

[14] `http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html`
[15] `http://www.freedesktop.org/wiki/Software/pkg-config`
[16] `http://www.steganosaur.us`
[17] `http://www.jrsoftware.org/isinfo.php`

that the user is running a compatible version of the Java Runtime Environment. The wrapper executable of our GUI is provided using another open source tool – `launch4j`[18]

Mac

Windows is somewhat unusual in the sense that it prefers all programs to be installed in a "Program Files" directory by a dedicated installer. With Mac OS however, applications are not "installed" as such, they generally just reside in an application bundle. Therefore our Java GUI Jar, and our command line executable are shipped for Mac OS inside an application bundle that is created by Apple's Jar Bundler[19]. Our application can be launched simply by opening the application bundle.

It is worth noting, that the latest version of Mac OS (which our solution supports) is provided with a compatible version of the Java Runtime Environment, therefore we do not need to explicitly check for the correct version of the Java Runtime Environment.

Linux

Linux has some similarities with Mac OS in the sense that it does not need applications to be installed to specific directories in its system, therefore our solution will be able to run "as is" provided that the user has a compatible version of the Java Runtime Environment.

As there are various linux distributions we cannot guarantee that the user will have a compatible version of the Java Runtime Environment installed, so we will need to check for this, and if they do not we should tell them this. After analysing other Java applications we have decided that our system will implement a bash script that checks for a compatible version of the Java Runtime Environment before attempting to launch the Jar for the GUI. The approach of using a bash script to launch the Jar is a technique that has been widely adopted by a variety of applications.

4.4 Chapter Summary

In this chapter we have discussed the design and implementation of our system. We have discussed at some length how the implementation of the system had to be adapted to overcome several challenges.

Implementing our system took several months, and a substantial portion of this time was focussed on the problems associated with modifying the motion vectors of a macro-block. As a result this limited the amount of time that we were able to spend experimenting and implementing different steganographic techniques.

Despite the unforeseen challenges that we have encountered whilst implementing our

[18] `http://launch4j.sourceforge.net/`

[19] `https://developer.apple.com/library/mac/documentation/Java/Conceptual/Java14Development/02-JavaDevTools/JavaDevTools.html`

system, we have been able to address the vast majority of our requirements.

Chapter 5: Testing

In Section 3.4 we outlined our testing strategy which focused predominantly on using unit testing to test the individual components that comprise our system. The entire system, including aspects that cannot be unit tested have been evaluated via comprehensive systems testing. Careful consideration has been given to the design and implementation of our system so that unit testing can be used as an effective testing strategy. The AES cryptosystem, transcoder, encoder, decoder and steganalysis tools are all individual components that can be tested independently before they are combined into a wider system.

Even though considerable care and consideration has been invested in producing a testable and accurate system we cannot guarantee that it is free from bugs or errors for two main reasons. Firstly, a video file can have many different configurations from stream data (such as the video resolution, bit rate, etc.) right through to the properties of each video frame (which are vast and variable). We therefore acknowledge that we have not been able to test our system for all possible configurations and audio/video parameter combinations. Secondly, the amount of systems and user acceptance testing that could be performed was limited by the amount of remaining time for this project, therefore preventing us from performing the extensive testing that we had originally planned for.

The process of testing our system has also been complicated by the fact that our solution is designed to be cross-platform and as a result, each element of our testing strategy should be performed on each operating system (Linux, Mac OS and Windows).

5.1 Unit Testing

In Chapter 4 we identified CUnit as our unit testing framework. Where possible, thorough unit testing has been implemented; however, unit testing is unable to verify the wider system, or the combination of component parts. Aspects of our system, such as the AES cryptosystem are entirely unit testable. The cryptosystem implemented in our system is entirely self-contained and strictly follows the FIPS-197 standard [Nat01], therefore the entire cryptosystem module was subjected to unit testing to verify that it met the specifications defined by this standard. NIST Special Publication 800-38A [Dwo01] defines encryption and decryption test vectors for all of the block modes and key sizes that our cryptosystem supports. These test vectors are therefore used to verify the performance and accuracy of the overall cryptosystem.

Our unit tests are divided into suites, one for each unit testable subsystem (AES cryptosystem, encoder and decoder). Unfortunately the transcode subsystem and aspects of the encoder and decoder are not unit testable. The results of our unit testing are summarised in figure 5.1.

Type	Total	Run	Passed	Failed	Inactive
suites	4	4	n/a	0	0
tests	38	38	38	0	0
asserts	572	572	572	0	n/a

Figure 5.1: CUnit summary – unit testing results

5.2 System Testing

The command line tool and GUI for our system were iteratively developed in parallel, which enabled us to exploit the graphical steganalysis tools of the GUI to help test the encoding and decoding modules.

For the most part system testing of the command line tool involved running it with different inputs and inspecting the output video file. For evaluating the process of encoding data in a video file we were able to use MPlayer[1] and our own steganalysis tools. MPlayer was primarily used to verify the accuracy of our steganographic tools as it allows for the analysis of motion vectors.

The suite of system tests outlined in tables 5.1 and 5.2 were used to test our finished system.[2]

[1] `http://www.mplayerhq.hu`

[2] The system testing grid in tables 5.1 and 5.2 was carried out on Linux, Mac and Windows, and the results were stored in separate test grids. As the results of the system testing were identical across all operating systems we have only include one system test grid to save on space.

Test	Expected Result	Actual Result
Command Line Tool		
Help system displays command list	`-h` Lists all commands	`-h` Lists all commands
Help system provides details of how to use commands	running `-h <command>` returns help instructions specific for that command	running `-h <command>` returns help instructions specific for that command
Decode message using first_mb_x method and no password	Message successfully decoded from video	Message successfully decoded from video
Decode message using first_mb_x method and correct password	Message successfully decoded from video	Message successfully decoded from video
Decode message using first_mb_x method and incorrect password	Message is unsuccessfully retrieved from video	Message is unsuccessfully retrieved from video
Decode message using first_mb_y method and no password	Message successfully decoded from video	Message successfully decoded from video
Decode message using first_mb_y method and correct password	Message successfully decoded from video	Message successfully decoded from video
Decode message using first_mb_y method and incorrect password	Message is unsuccessfully retrieved from video	Message is unsuccessfully retrieved from video
Encode message using first_mb_x method and a password	Message is successfully embedded	Message is successfully embedded
Encode message using first_mb_x method and no password	Message is successfully embedded	Message is successfully embedded
`-cryptographic-algorithms` command	Should output the names of the supporting cryptographic algorithms.	Should output the names of the supporting cryptographic algorithms.
Encryption and decryption test vectors	The test vectors produced by the `-encrypt-test-vector` and `-decrypt-test-vector` match the vectors stated in [Dwo01]	The test vectors produced by the `-encrypt-test-vector` and `-decrypt-test-vector` match the vectors stated in [Dwo01]
Meta data accuracy	Meta data outputted should match the meta data derived using VLC media player	Meta data outputted should match the meta data derived using VLC media player
Motion vector dump	Output the frame type and an accurate matrix of motion vectors for each macroblock in a frame	Output the frame type and an accurate matrix of motion vectors for each macroblock in a frame
`-schemes` command	Should output first_mb_x and first_mb_y as the available encoding schemes.	Should output first_mb_x and first_mb_y as the available encoding schemes.
Transcode Mechanism	Should produce an output video file capable of playback and matches the quality of the input	Should produce an output video file capable of playback and matches the quality of the input
`-unit-test` command	Running the `-unit-test` command should execute all of the unit test suites	Running the `-unit-test` command should execute all of the unit test suites
Video stream overview	Running `-vs-overview` outputs a summary of the frame type information for each frame in the input video.	Running `-vs-overview` outputs a summary of the frame type information for each frame in the input video.

Table 5.1: System test grid – command line tool

Test	Expected Result	Actual Result
Graphical User Interface		
Encode	Encodes a video using the command line tool and reports the progress via a progress bar	Encodes a video using the command line tool and reports the progress via a progress bar
Decode	Decodes a video using the command line tool and reports the progress via a progress bar	Decodes a video using the command line tool and reports the progress via a progress bar
Cryptography Test Vectors	Parses parameters to the command line tool and displays the resulting vector	Parses parameters to the command line tool and displays the resulting vector
Unit Test	Runs `-unit-test` command of the command line tool and displays the output text	Runs `-unit-test` command of the command line tool and displays the output text
Transcode	Transcodes a video using the command line tool and reports the progress via a progress bar	Transcodes a video using the command line tool and reports the progress via a progress bar
Execute	Runs the command line tool with the specified parameters and displays the output	Runs the command line tool with the specified parameters and displays the output
Video Player – Open File	Plays the specified video file	Plays the specified video file
Video Player – Play	Synchronised playback of audio and video streams	Synchronised playback of audio and video streams
Video Player – Stop	Stops playback of audio and video streams	Stops playback of audio and video streams
Meta Data	Runs `-meta-data` command of the command line tool and displays the output text	Runs `-meta-data` command of the command line tool and displays the output text
Visual Comparison	Display selected frames from the specified video	Display selected frames from the specified video
Motion Vector Comparison	Using the command lien tool display motion vectors for the specified videos and frames and colour the differences	Using the command lien tool display motion vectors for the specified videos and frames and colour the differences
Video stream overview	Runs `-vs-overview` command of the command line tool and displays the output text	Runs `-vs-overview` command of the command line tool and displays the output text
Check for updates (update required)	User is told that a new version of the system is available and asks them if they want to go to the website to download.	User is told that a new version of the system is available and asks them if they want to go to the website to download.
Check for updates (up-to-date)	User is told that there system is up-to-date	User is told that there system is up-to-date

Table 5.2: System test grid – GUI

5.3 Video Testing

The video files produced by our system also required thorough testing. It is important that they are valid video files that are capable of playback. Videoplay was evaluated using various third-party media players, including VLC[3], Windows Media Player[4] and QuickTime[5]. Our tests showed that different media players can have different levels of

[3] `http://www.videolan.org/vlc/`
[4] `http://windows.microsoft.com/en-GB/windows/download-windows-media-player`
[5] `http://www.apple.com/uk/quicktime/`

tolerance for faults or imperfections in a media file, it is for this reason that we have tested our video across a variety of media players, and on different operating systems.

We have tested our video files by allowing complete playback of the media from start to finish – this tests normal sequential decoding. Candidate video files are then further tested by seeking to random positions in the video file – this tests non-sequential decoding. If the keyframe index of a video file is incomplete, inaccurate or corrupt this will disrupt the process of seeking to a position in the stream, because you can only directly seek to keyframes (I-frames). The exact seek position is achieved by seeking to the closest I-frame and then interpolating any P- and B- frames between the I-frame location and the requested seek position.

Attention is also afforded to the synchronisation of audio and video, and the file size. It goes without saying that the streams of our output file should synchronise in the same fashion of those in the input file. From our experience we have also learned that a significantly inflated output file size can be a first indicator of erroneous video coding.

5.4 User Acceptance Testing

During this project we were asked to provide a guest lecture for a range of students and researchers within the Department of Computer Science; at this lecture we discussed video steganography and our system at length, as well as providing live demonstrations of our work. In addition to this, we also made our software available to the audience for further user acceptance testing and feedback.

During the lecture we surveyed 12 members of the audience. According to their feedback our system was clean, easy and intuitive to use. The only suggestion that we received was that "a help section may come in handy". Although our command line tool does provide help functionality, our GUI tool does not. Unfortunately due to time constraints we were unable to act on this suggestion.

We also attempted to determine whether the audience was able to detect a video that had data hidden in it. We showed them three identical videos simultaneously, one of which had data embedded using one of our techniques. Although the results of the survey are interesting we believe them to be inconclusive. Our survey showed that 50% correctly identified the video with hidden data and 50% said that none of the videos contained hidden data. Unfortunately we did not collect the survey results until the end of the lecture, but we did reveal the answer during the talks. As a result, some of the participants may have written down the answer once we told them. It seems unusual that all of the participants would either correctly identify the video, or else indicate that nothing was embedded at all.

5.5 Chapter Summary

We have used a variety of testing and evaluation techniques to verify the performance and accuracy of the software solution that we have produced. We have combined unit testing, system testing and user acceptance testing to produce a thorough and comprehensive testing strategy. The user acceptance testing that we have carried out suggested that the solution we have produced is user friendly, but the feedback relating to the subtlety of the embedding technique seems to be somewhat inconclusive or unreliable. The testing

that we have carried out indicates that our system can accurately perform to the level of functionality that we have indicated.

Chapter 6: Evaluation

During the course of this project we have developed a video steganography tool that uses modern motion vector techniques to embed and extract data in H.264 (and MPEG4) which is the most commonly used online video format. The system that we have produced has proven to be successful in hiding data in video files by using motion vector manipulation. Although time constraints prevented us from exploring more steganographic schemes, our project concludes at the point where the further development of these techniques is possible.

6.1 Deliverables and Implementation

Throughout the course of this project we have implemented a steganographic system that is capable of hiding data in H.264 and MPEG4 video files using two different methods of embedding. Our tools are capable of taking an object file (of any data format) and encrypting (with 256-bit AES encryption) and embedding the data in a video so that the resultant video is indistinguishable from the original container file and is capable of normal video playback.

Several steganalysis tools were also developed that proved useful in evaluating and testing our system. These tools allow us to analyse how we embed the data and the visual similarities and differences between video frames.

In our requirements and analysis we categoried the requirements of our system as being "mandatory", "desirable" or "optional". We have successfully implemented all of our mandatory requirements and most of the desirable requirements. Due to time constraints we were not able to complete the implementation of the desirable requirements and only a few of the optional requirements have been implemented.

At the start of this project the intention was to research and develop numerous schemes for embedding and extracting data covertly from video files. Our original goal was heavily ambitious and did not fully recognise the complexities associated with manipulating video data. Furthermore, out video manipulations had to be lossless and recoverable. Although we have not managed to implement as many steganographic algorithms as we had originally intended, we have during the lifecycle of this project developed two methods for embedding and extracting data from a video file.

Our original plan (as defined in our Survey and Analysis document) detailed a rigid six-iteration steganography phase in which a new steganographic scheme would be implemented and analysed for each iteration. This was unbelievably optimistic, although at the time we believed significant contingency time was built into all of the main tasks (such as transcoding, cryptography, etc.). With hindsight it is quite clear that we did not properly estimate and forecast the challenges that we would encounter on this project and the lengthy process involved in resolving these. Developing the system so that it was capable

of hiding data was undoubtably one of the most time consuming aspects of the system, that took far longer than we originally anticipated. Having undertaken this project it would appear that video steganography is a highly specialised field. Before embarking on this project we had no prior knowledge of steganography or video coding/processing which at points caused steep and dramatic learning curves.

Our final solution uses motion vector based approaches that are capable of hiding data in H.264 and MPEG4 video files. As part of the testing and evaluation of our system we have made binary executables of our software available for public download. To the best of our knowledge, our tool is the only video steganography system capable of hiding data in this manner that has been made publicly available via the internet.

6.2 Results and Findings

Our main goal at the outset was to develop and research techniques for embedding data using motion vector based techniques. For the schemes that we have developed this has proven to be a success, although we have noted a couple of limitations with our work. At the start of this project we believed that it would be possible to embed data in any motion vector of a P- and B- frame. For the most part this is true, however it is possible for a macroblock to be coded as having no motion vector. This was not a concept that we were familiar with. Coding a macroblock with no motion vector is different to coding a motion vector with a resultant magnitude of zero.

From an implementation stance, FFmpeg would use the `mb_type` attribute of the `AVFrame` struct to indicate the type of macroblock (and subsequently whether a motion vector is coded or not). Prior to discovering this our system would modify the `motion_val` attribute of `AVFrame`, but, then the coding process of FFmpeg would ignore the motion vector when the macroblock type indicated that this was appropriate. We mitigated this problem by inspecting the type of each macroblock prior to encoding.

Another limitation of our system is the inability to determine steganographic capacity prior to encoding. Motion vectors describe the spatial translation of a block of pixels between frames, whence modifying the motion vectors in one frame will have an impact on other motion vectors. This relationship means that when coding/manipulating one macroblock, another macroblock may be changed as a result of this action so that it encodes no motion vector. This type of action causes the number of encodable macroblocks to change depending on the object that is to be embedded.

The number of available macroblocks can also vary due to key frame positions or GOP sizes. Whilst it is good practice to have I-frames at regular intervals[1], it is possible for a video to have irregular and inconsistent GOP sizes. During the coding process our system will default to a GOP size of 12 if a regular GOP size cannot be detected. As a result of this action B- or P- frames will change to I-frames which have no macroblocks, therefore drastically changing the number of available macroblocks. Whilst we could change our system to preserve the I-frame position of the input video a regular GOP size provides better error correction.

For these reasons it is not possible to detect or the steganographic capacity of a video. If our system is unsuccessful in embedding the entire object, we inform the user of this, and the size of data that was successfully embedded. We indicate to the end user that this is an approximation of the steganographic capacity of the video.

[1] GOP sizes of 12 occur frequently across video formats.

Figure 6.1: Input Video – frame 600

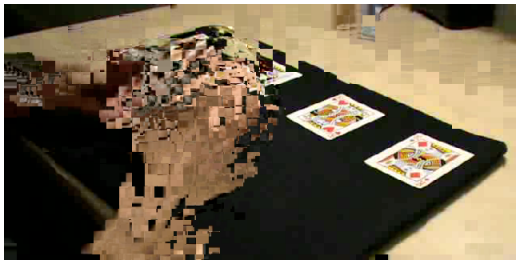


Figure 6.2: Inverted motion vectors – frame 600

Figure 6.3: With "First Macroblock X" technique applied – frame 600

Figure 6.4: With "First Macroblock Y" technique applied – frame 600

Our system is capable of manipulating motion vector values and hiding data using one of two methods:

- **First Macroblock X**
  This method hides data by modifying the LSB of the X-component of the motion vector in the first macroblock.
- **First Macroblock Y**
  This method hides data by modifying the LSB of the Y-component of the motion vector in the first macroblock.

In figures 6.1 and 6.2 we illustrate how our system is capable of modifying motion vector values quite significantly. In this example the motion vectors for every macroblock are inverted. This illustrates how motion vector modification can produce significant and noticeable artefacts if the manipulation is not performed with care.

Figures 6.3 and 6.4 show the same video frame (frame 600), this time with data embedded in the frame using the different techniques. In both instance the first macroblock of the frame (top-left corner) is the area in which the manipulation is occurring. We believe that this technique is subtle, although the manipulation occurs in the same corner of the frame, we believe that the LSB portion is small enough that it does not cause obvious or noticeable artefacts.

In an attempt to verify this we conducted a survey with 12 people during a lecture. We showed them three identical videos simultaneously one of which had data embedded using the "First Macroblock X" technique. Although the results of the survey are interesting we believe them to be inconclusive, as explained in Section 5.4.

From a pure steganalysis point of view, it is not easy to spot changes that have been made to the motion vectors of the frame. In Figure 6.5 we analyse the motion vector values in frame 600 of the original and "First Macroblock X" videos. It is worth noting that in

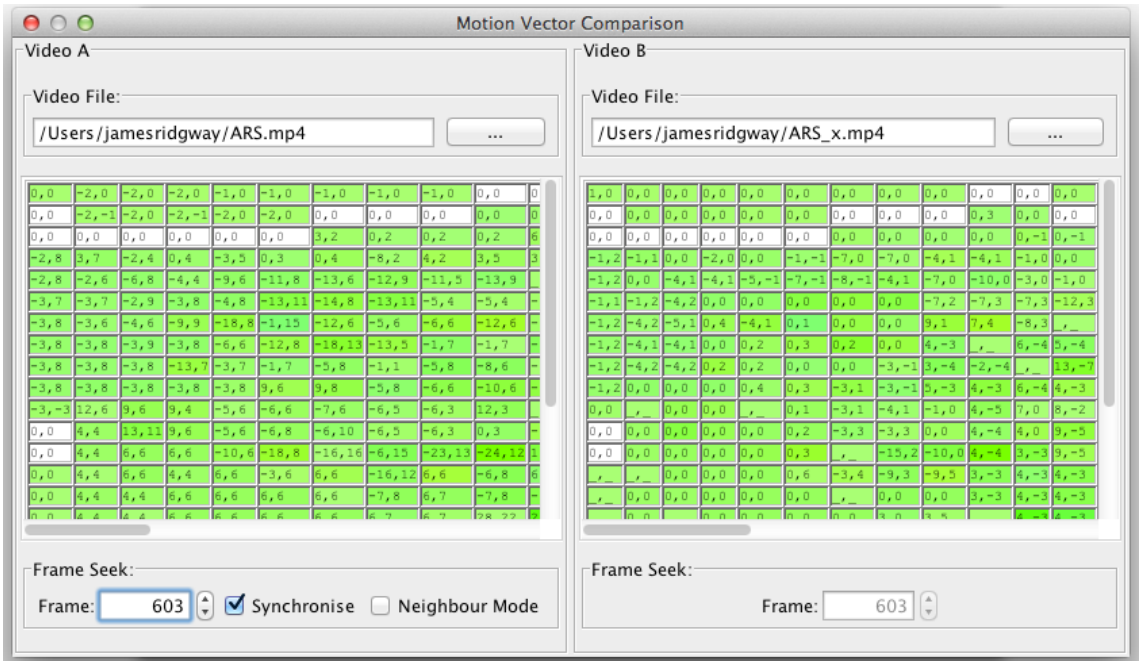


Figure 6.5: Motion vector comparison of original and "First Macroblock Y" video – frame 600

this frame the X-component is not modified, but nonetheless there is significant fluctuation between the motion vector values shown for Video A and Video B. This fluctuation is caused by the lossy nature of H.264/MPEG4 video coding, which means the simple process of transcoding a video will cause motion vector values to fluctuate. These results and observations also hold for frames where the X-component has been modified

The original goal of this project was to experiment with different techniques for hiding information in video files. This project has succeeded in implementing two such strategies. Although time constraints meant that we could not further explore other schemes, and were unable to implement features such as an independent steganographic and cryptographic keys, the overall goal of hiding data by manipulating motion vectors proved to be a success.

6.3 Further Work

With the luxury of additional time this project could be expanded to explore other, more complex, steganographic schemes. Now that the concept of hiding information in the motion vectors of an H.264/MPEG4 video has been proven, we could experiment with methods that have high steganographic capacity or are difficult to detect, and those which achieve a middle ground between the two criteria. The steganographic schemes that have been implemented thus far only modify the motion vectors of a single macroblock per frame. We believe that with a small amount of additional work we could produce a system that is capable of embedding in all the macroblocks of a frame, whilst producing minimal distortion to the original image.

The concept works by hiding information in a frame just before an I-frame. This frame will be shown so briefly before the I-frame is displayed that any distortion should not be easily noticeable. It is hoped that this proposed technique will be less noticeable than those

Figure 6.6: Embedding in every macroblock of every frame. Frame 606 (precedes I-frame).

Figure 6.7: Embedding in every macroblock of every frame. Frame 606 (I-frame).

Figure 6.8: Embedding in every macroblock of every frame. Frame 607 (post I-frame).

we have implemented because we only modify a single frame in each GOP, whereas in our implemented techniques we are modifying the same macroblock in consecutive frames.

We conducted an experiment where we attempted to manipulate the LSB of every macroblock in every frame. This proved to be very unsuccessful because the changes to the motion vectors aggregated within a GOP. In our experiment we used a GOP size of 12 frames. Figure 6.6 shows the frame immediately before an I-frame (figure 6.7), and figure 6.8 shows the frame immediately after the I-frame. Figures 6.6 and 6.8 both have data hidden in them, but because figure 6.6 is the final frame in its GOP the manipulations made to the motion vectors in the preceding frames have influenced those that follow. Frame 6.8 shows that it is possible to modify all the macroblocks in a frame with minimal distortion.

One technique that we would therefore propose as further work would involve testing the proposal of embedding data in the motion vectors of frames that immediately precede an I-frame.

There are numerous options for further research into the use of video steganography techniques and software. Research could be continued into the motion vector approach to determine whether it is possible to generalise our approach for all motion vector based codecs.

Unfortunately, due to time constraints, we were unable to develop a scheme capable of embedding in both audio and video. Although this was one of our lower priority requirements, this is an area of steganography that is yet to be explored, and how significantly this can impact steganographic capacity. To the best of our knowledge there is no mention of a steganography technique in the literature that is capable of embedding in both the audio and video stream of files.

6.4 Chapter Summary

From the outset we believed that this project would be complicated and challenging to undertake, but even with this in mind we underestimated quite how difficult this would be. As a result a large portion of time was consumed developing a transcoding mechanism and an appropriate process for manipulating motion vectors. Despite the setbacks that we have encountered we have been able to produce a system that achieves most of the goals that we set out to achieve. The requirements that we have not implemented have been unattainable due to time constraints as opposed to implausibility.

Chapter 7: Conclusion

From the outset our project has been undertaken to explore the possibility of embedding data using motion vector techniques in appropriate file formats.

Whilst undertaking the literature survey, we also started to undergo preliminary research into image and audio steganography techniques and video manipulation (see Appendix A). The knowledge that we gained from this preliminary research helped us to understand how steganographic techniques work with compressed media formats and the difficulties of manipulating video data.

The literature survey revealed numerous video steganography sources, but most of these sources were concerned with slightly older DCT based techniques. Very few papers discussed steganographic techniques that were applicable to modern video file formats such as H.264. In all instances, the literature described how to apply the technique at a very high level, seeming to work on the understanding that you knew how video coding works and how to develop a system capable of performing video manipulations. As a result we had to teach ourselves how to built a video transcoder that was capable of making modifications to video frame data before it was coded into a video packet.

Before we were able to achieve this goal we had several false starts. We first attempted to use Java and Xuggler to hide data in video. However, we found that Xuggler would not allow us to perform the low level operations that were needed in order to change the motion vectors of the video file.

Our next attempt involved switching to using C and FFmpeg. Our original C code used FFmpeg as a linked library and we attempted to modify the `AVFrame`s motion vector attributes before they were passed to the encoder methods, but unfortunately it transpired that we would need to modify the behaviour of the FFmpeg encoder in order to apply the manipulations we needed, As a result, our final solution incorporates a modified version of the FFmpeg source code.

Whilst we were trying to ascertain a transcode mechanism that would allow us to modify the motion vectors of a video, our designs and implementation were being iteratively updated. Each iteration and alteration to the original design was a result of acquiring more knowledge about how video coding and video manipulation works. Whilst there is information in the literature on video codecs and how they work, there is very little information on how to build transcoding systems.

Even though FFmpeg is the most comprehensive video manipulation library current

under active development, there are almost no examples of how to use the API afforded by the library. Only a handful of examples are provided in the API documentation, but even though a few of these examples contained useful snippets of code that showed us how to use various function of the API, we had to develop (through a lot of experimentation) our own code completely from scratch.

Attempting to learn how to use a vast and complex codebase such as FFmpeg, which is written in C – a language of which we had no prior experience – was certainly challenging. Undoubtably the learning curve that was introduced here detracted from the time that we could have spent working on steganographic techniques. According to the original plan in our survey and analysis we would have undertaken a six-iteration steganography and steganalysis phase in which a single steganographic scheme would have been produced per iteration. This time frame proved to be completely unrealistic, but nonetheless we conclude our project with two steganographic schemes that are capable of hiding data in H.264 and MPEG4 video files.

We have successfully developed a cross-platform system that is capable of hiding data in H.264 and MPEG4 video files. Although due to time constraints we were unable to implement all of our desirable and optional requirements, we feel that with a small amount of further work we would be able to address these issues. Our exploration of the processes involved in manipulating video data has prompted numerous ideas for different steganographic schemes and there is a lot of potential for developing this system further and expanding this research into the field of video steganography.

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.

[AT90] C. M. Adams and S. E. Tavares. The structured design of cryptographically good s-boxes. *Journal of Cryptology*, 3(1):27–42, 1990.

[AWSZ05] I. Ahmad, X. Wei, Y. Sun, and Y. Zhang. Video transcoding: An overview of various techniques and research issues. *IEEE Transactions on Multimedia*, 7(5):793–804, October 2005.

[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.

[BF11] A. A. Bruen and M. A. Forcinito. *Cryptography, Information Theory, and Error-Correction: A Handbook for the 21st Century.* Wiley, 2011.

[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.

[Cas10] L. Case. All about video codecs and containers. `http://www.pcworld.com/article/213612/all_about_video_codecs_and_containers.html`, 2010. Date Accessed: 9 April 2013.

[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, October 1999.

[Coc73] C. C. Cocks. A note on non-secure encryption, 1973.

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

[Con08] Conceiva Pty. Ltd. White paper: Download managers – a better downloading experience. `http://www.conceiva.com/products/downloadstudio/WhitePaper-DownloadManager.pdf`, 2008. Accessed: 9 April 2013.

[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.

[DaC] DaCast. Streaming service for flash, rtmp, h.264 & vp6. `http://www.dacast.com/flash-rtmp-h264-vp6-streaming.html`. Date Accessed: 9 April 2013.

[Dav97] D. Davies. A brief history of cryptography. *Information Security Technical Report*, 2(2):14–17, 1997.

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[DHC10] R. C. Dodge Jr., T. Holz, and A. Chuvakin. Advanced attacker detection and understanding with emerging honeynet technologies. In *Wiley Handbook of Science and Technology for Homeland Security*, page 984. John Wiley & Sons, Inc., 2010.

[Dip08] B. Dipert. Online video content distribution: Sony's playstation 3 enters the ring (albeit with a sound-hampered hand tied behind its back). `http://www.edn.com/electronics-blogs/brians-brain/4305143/Online-Video-Content-Distribution-Sony-s-PlayStation-3-Enters-The-Ring-Albeit-With-A-Sound-Hampered-Hand-Tied-Behind-Its-Back-`, 2008. Date Accessed: 9 April 2013.

[DK07] H. Delfs and H. Knebl. *Introduction to Cryptography: Principles and Applications*. Springer, 2007.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[Dwo01] M. Dworkin. Nist special publication 800-38a: Recommendation for block cipher modes of operation. `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`, 2001. Date Accessed: 9 April 2013.

[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.

[Ell70] J. H. Ellis. *The possibility of non-secret digital encryption.* Government Communication Headquarters (GCHQ), 1970. `http://cryptocellar.web.cern.ch/cryptocellar/cesg/possnse.pdf`. Accessed: 9 April 2013.

[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.

[Fei73] H. Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, May 1973.

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

[FNS75] H. Feistel, W. A. Notz, and J. L. Smith. Some cryptographic techniques for machine-to-machine data communications. *Proceedings of the IEEE*, 63(11):1545–1554, November 1975.

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

[FSK10] N. Ferguson, B. Schneier, and T. Kohno. Cryptography engineering: Design principles and practical applications, 2010.

[GARR05] B. Girod, A. M. Aaron, S. Rane, and D. Rebollo-Monedero. Distributed video coding. *Proceedings of the IEEE*, 93(1):71–83, January 2005.

[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. Accessed: 9 April 2013.

[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.

[KD79] J. B. Kam and G. I. Davida. Structured design of substitution-permutation encryption networks. *Computers, IEEE Transactions on Computers*, C-28(10):747–753, October 1979.

[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. Podgórski. Log-Laplace distributions. *Internat. Math. J.*, 3:467–495, 2003.

[Le 91] D. Le Gall. Mpeg: a video compression standard for multimedia applications. *Commun. ACM*, 34(4):46–58, April 1991.

[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.

[Mar03] K. Martins. How video sharing has become more popular with adults and adolescents alike. `http://cs36.com/video-sharing/`, 2003. Date Accessed: 9 April 2013.

[Mit00] S. K. Mitra. *Digital Signal Processing: A Computer-Based Approach.* McGraw-Hill Inc., 2000.

[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.

[MOVR96] A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone, and R. L. Rivest. Handbook of applied cryptography, 1996.

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

[Nat01] National Institute of Science and Technology. Federal information processing standards publication 197. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2001. Accessed: 9 April 2013.

[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, October 2004.

[O'C95] L. O'Connor. On the distribution of characteristics in bijective mappings. *Journal of Cryptology*, 8(2):67–86, 1995.

[ÓDB96] 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, August 1996.

[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.

[Ric08] I. E. G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia*. John Wiley & Songs, 2008.

[Ros08] A. Rose. BBC iPlayer Goes H.264. `http://www.bbc.co.uk/blogs/bbcinternet/2008/08/bbc_iplayer_goes_h264.html`, 2008. Date Accessed: 9 April 2013.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[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.

[Sch10] E. Schonfeld. H.264 already wonmakes up 66 percent of web videos. `http://techcrunch.com/2010/05/01/h-264-66-percent-web-video/`, 2010. Date Accessed: 9 April 2013.

[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.

[Smi71] J. L. Smith. The design of lucifer, a cryptographic device for data communications. Research Report RC3326, IBM, 1971.

[Sti02] D. R. Stinson. *Cryptography Theory and Practice*. Chapman & Hall/CRC, 2002.

[Stu08] S. Sturgeon. Showdown: Apple tv vs. vudu. `http://www.hdtvmagazine.com/columns/2008/02/showdown_apple_tv_vs_vudu.php`, 2008. Date Accessed: 9 April 2013.

[Tek00] Tektronix. A guide to mpeg fundamentals and protocol analysis. `http://www.img.lx.it.pt/~fp/cav/Additional_material/MPEG2_overview.pdf`, 2000. Date Accessed: 9 April 2013.

[Vim] Vimeo. Video compression guidelines. `http://vimeo.com/help/compression`. Date Accessed: 9 April 2013.

[Wan09] J. Wang. *Computer Network Security*. Springer London, Limited, 2009.

[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.

[Whi90] D. Whitehead. *How to survive under siege*. Clarendon ancient history series. Clarendon Press, 1990.

[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 *ICICIC '06. First International Conference on Innovative Computing, Information and Control, Aug 30 – Sep 1, 2006.*, volume 1, pages 269–272, 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.

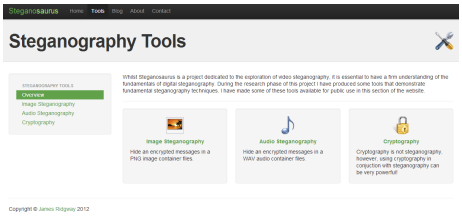
Appendix A: Preliminary Research

A.1 Steganosaurus

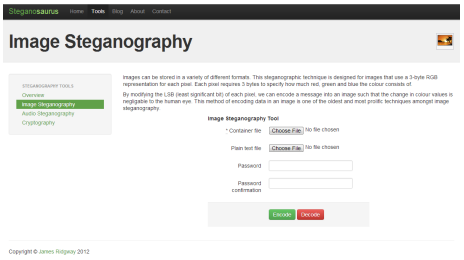
"Steganosaurus" is the nickname given to this project. From the outset a website (`http://www.steganosaur.us`) was setup to document the research and developments of this project. Through the blog we recorded all progress, set backs, discoveries and additional details as frequently as possible.

Figure A.1: Steganosaur.us - Homepage

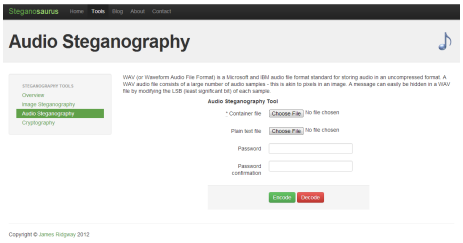
The *Tools* section of our website allows visitors to use some of the steganography and cryptography tools that we produced during the preliminary research phase of this project.



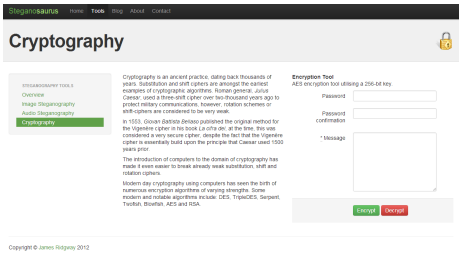
(a) Steganosaur.us - Tools section



(b) Steganosaur.us - Audio steganography tool



(c) Steganosaur.us - Image steganography tool



(d) Steganosaur.us - Cryptography tool

Figure A.2: Steganosaur.us – Website Sections

A.2 Preliminary Research

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.2.1 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 substitution-based steganographic technique for audio files.

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 bytes 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.3 Early Steganography System

Before arriving at our final steganography system (which is written in C, with a Java GUI), we originally attempted to develop the system entirely in Java. Whilst we were building this early Java based steganography system we also included some of our preliminary research tools, such as the image and audio steganography tools and basic image steganalysis tool.

The figures in this section show the level of functionality implemented in 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 A.3.2 for further details).

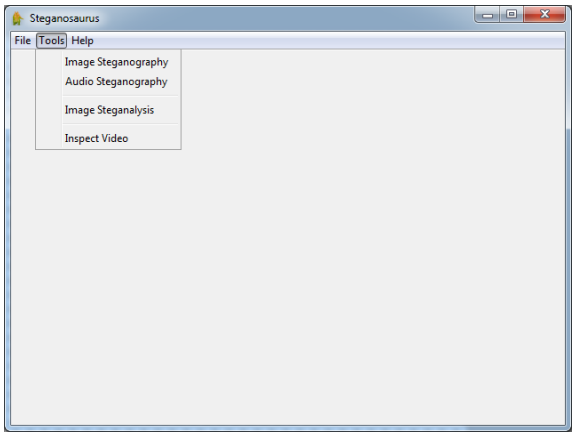
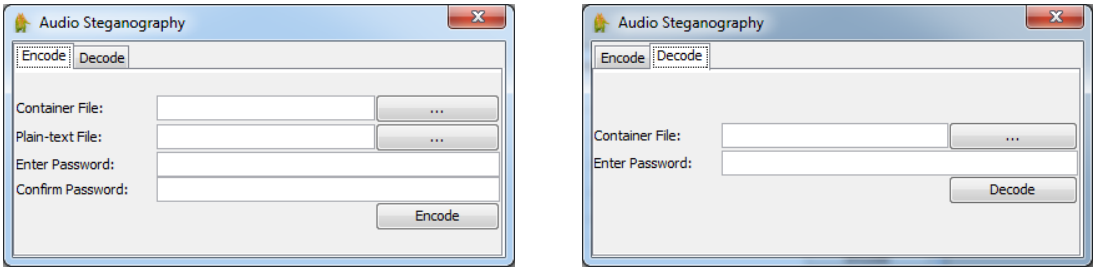


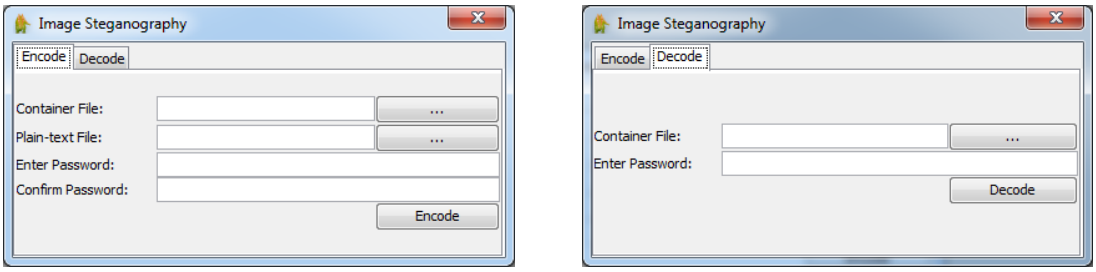
Figure A.3: Main interface

From the main interface, the following dialogs were accessible:



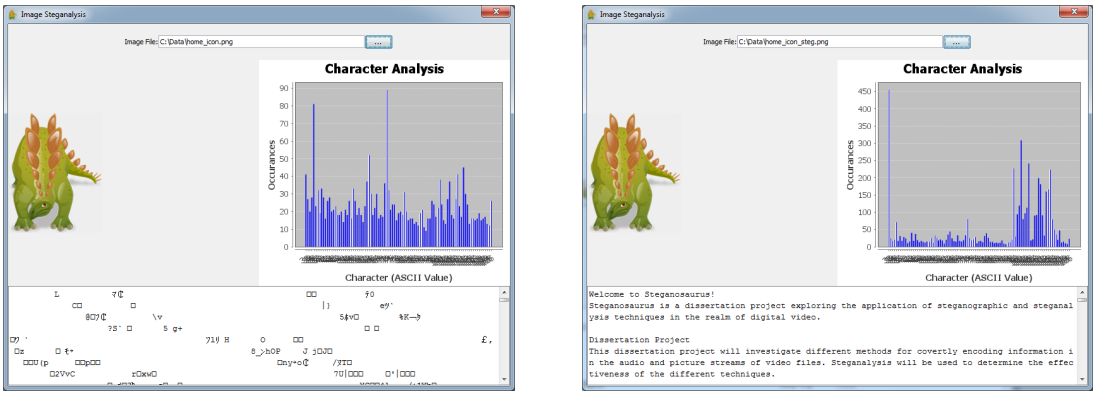
(a) Encoding Mode. (b) Decoding Mode.

Figure A.4: Audio Steganography Tool.



(a) Encoding Mode. (b) Decoding Mode.

Figure A.5: Image Steganography Tool.



(a) Steganalysis screen showing when analysing an image *without* data embedded. The textbox contains the LSB string.

(b) Steganalysis screen showing when analysing an image *with* data embedded. The textbox contains the LSB string.

Figure A.6: ASCII distribution comparison

A.3.1 Image Steganography and Steganalysis

It is worth taking a closer look at what our preliminary research demonstrated in figure A.6. Figure A.6 summarises the comparison of an image before and after LSB embedding.

(a) Graphic before data is embedded (b) Graphic after data is embedded

Figure A.7: Steganosaurus Graphic

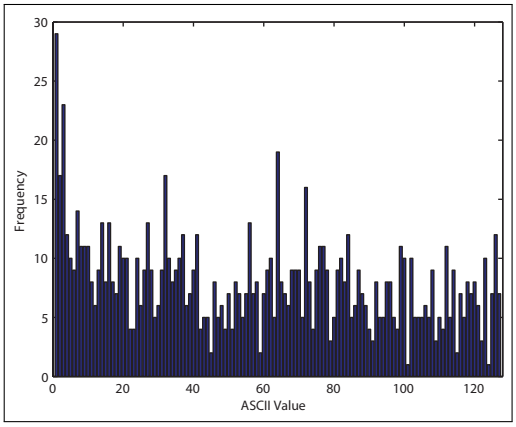


Figure A.8: ASCII distribution of LSB string after encrypted data is embedded in a PNG image.

Figures A.7a and A.7b show the original and manipulated image file respectively – note that there is no noticeable difference between the figure. 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 A.7a and A.7b respectively.

Further experimentation showed that by encrypting the data before embedding, the presence of embedded data is better disguised, as demonstrated in figure A.8. The distribution of ASCII values in figure A.8 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.

A.3.2 Early Video Manipulation

Our early work on video manipulation was started in Java using a library called Xuggler[1]. Xuggler is a Java API for video that acts as a wrapper for FFmpeg. After learning how to extract basic meta data from the video file, experimentation progressed to video manipulation. Initially, progress was good; figures A.9a, A.9b and A.9c show frames from an inverted 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

[1] `http://www.xuggle.com/xuggler`

(a) Original video frame (b) Visually watermarked video frame (c) Inverted video frame

Figure A.9: Video frame manipulation using Xuggler

image within the spatial domain. After some extensive research it was concluded that Xuggler can provide an excellent range of high-level functionality, however this project requires manipulating video data at a much lower-level.

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 A.9c).

- `us.steganosaur.VideoWatermarker`
  This class applies an image watermark to the bottom right of a video (see figure A.9b).

A.3.3 Deviation from Java

As we mentioned in Section A.3.2, we had to move away from using Java as we needed access to the lower level functionality of FFmpeg. As a result we switch to using C as we could natively interface with FFmpeg and could modify the source code, which we later found we had to do. This proved to be a substantial set back, not only did we have no prior knowledge of C, but we had to interface with and modify a complex C library

consisting of over 650, 000 lines of code.

Appendix B: Advanced Encryption Standard

B.1 The Advanced Encryption Standard

The Advanced Encryption Standard (AES) is the current de facto encryption algorithm for securing sensitive information, and is used by governments and militaries across the world.

In January 1997 the National Institute of Standards and Technology (NIST), a subdivision of the U.S Department of Commerce, start looking for a replacement to the Data Encryption Standard (DES) which had been the encryption standard for nearly two decades. DES is now considered to be too insecure for many applications.

Candidate algorithms were submitted to NIST for consideration as alternatives to DES. All of the algorithms submitted were subjected to review and analysis by NIST and general public. Rijndael (pronounced "rain dahl") was finally chosen as the AES out of the final 15 candidate algorithms. The Rijndael algorithm was developed by Belgian cryptographers, Joan Daemen and Vincent Rijmen.

B.1.1 AES and Rijndael

Rijndael and AES are often used interchangeably, when they are technically different. Strictly speaking AES is an implementation of Rijndael. Rijndael can have any multiple of 32-bit block size and key size between 128- and 256-bits, AES has a strict block size of 128-bits and keys must be either 128-, 192- or 256- bits in length [DR02, p. 31]. The exact specification of AES is formalised in the Federal Information Processing Standard (FIPS) 197 [Nat01] which was released by the Secretary of Commerce on 6 December 2001.

B.1.2 Algorithm Overview

AES is a symmetric key block cipher. Symmetric key algorithms are encryption algorithms that use the same cryptographic key for the encryption and decryption process. Block ciphers and stream ciphers are the types of symmetric key algorithms [Sch96, p. 4]. A stream cipher will encrypt bits one at a time. On the other hand, a block cipher will read in a number of bits (called a block) and encrypt them as a single entity. If the plaintext bits that are being encrypted are less than the block size, the plaintext will be padded so that the plaintext matches the block size. AES always produces encrypted messages that are multiples of 128-bits because of the 128-bit block size. Although the algorithm uses a fixed block size it does accept a variety of key sizes: 128-, 192- and 256- bits. AES is based on a design principle called a substitution-permutation network (SPN). An SPN uses substitution boxes (S-Boxes) and permutation boxes (P-Boxes) to apply layers of

substitution and permutation to a block. S- and P- box transformation are often achieved using exclusive or and bitwise rotation [DR02][p. 77] [AT90, FNS75, KD79, O'C95].

During the encryption and decryption process repetitive transformation rounds are applied to a 4x4 column-major order matrix - known as a state matrix. The number of rounds of transformation repetitions that are applied to the state matrix are governed by the key size.

- 10 rounds for 128-bit keys.
- 12 rounds for 192-bit keys.
- 14 rounds for 256-bit keys.

Each round of encryption involves several processing steps which are used to transform the plaintext to ciphertext. Conversely, inverse processing steps exist to transform ciphertext back to plaintext during the decryption process.

The AES algorithm can be summaries into the following four major steps:

1. Key Expansion
2. Initial Round
   (a) `AddRoundKey`
3. Iterate Rounds
   (a) `SubBytes`
   (b) `ShiftRows`
   (c) `MixColumns`
   (d) `AddRoundKey`
4. Final Round
   (a) `SubBytes`
   (b) `ShiftRows`
   (c) `AddRoundKey`

B.2 Algorithm Processes

The definitions of the algorithm processes in the section are derived from the descriptions in [DR02].

B.2.1 Key Expansion

Key expansion derives round keys for the cipher key using the Rijndael Key Schedule. The Key Schedule expands a short key into a number of separate round keys. The key schedule algorithm uses a number of core operations: Rotate, Rcon and Rijndael S-Box.

Operations

Rotation

The rotation operation takes a 32-bit word and rotates it 8-bits to the left, for example, `1D 2C 3B 4A` becomes `2C 3B 4A 1D`.

Rcon

Rcon (or round constants) are values computed in GF(2^8), whereby:

$$Rcon(i) = x^{i-1} \bmod x^8 + x^4 + x^3 + x + 1 \tag{B.1}$$

with $x = 2$ and i starting at 1.

Note: this operation is performed as a polynomial in the finite field GF(2^8) and not as real integers. The example below shows how to calculate the $Rcon(9)$:

$$\begin{aligned} Rcon(9) &= x^{9-1} \bmod x^8 + x^4 + x^3 + x + 1 && \text{(B.2)} \\ &= x^8 \bmod x^8 + x^4 + x^3 + x + 1 && \text{(B.3)} \\ &= 100000000 \bmod 100011011 && \text{(B.4)} \\ &= 11011 = (27 \text{ in decimal}) && \text{(B.5)} \end{aligned}$$

Rijndael S-Box

The Rijndael S-Box is a lookup table of values (see B.2.2), which is used with a `SubWord` function that applies the S-Box to each of the 4-byte of the input word to produce an output word.

Key Schedule Pseudocode

The following pseudocode from FIPS PUB 197 [Nat01] outlines the operation of the Rijndael Key Schedule algorithm:

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
  word temp
  i = 0
  while (i < Nk)
    w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
    i = i+1
  end while
  i = Nk
  while (i < Nb * (Nr+1)]
    temp = w[i-1]
    if (i mod Nk = 0)
      temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
    else if (Nk > 6 and i mod Nk = 4)
      temp = SubWord(temp)
    end if
    w[i] = w[i-Nk] xor temp
    i = i + 1
  end while
end
```

B.2.2 `SubBytes` Step

`SubBytes` consists of applying an S-Box permutation to the bytes of the state matrix. This is the only non-linear transformation in the entire cipher. The S-Box used by AES was

designed to minimise the input-output correlation, and difference propagation probability. The S-Box is derived from the multiplicative inverse in GF(2^8), and is defined as:

$$g : a \to b = a^{-1} \tag{B.6}$$

B.2.3 `ShiftRows` Step

The `ShiftRows` step involves applying a left circular shift to each row of the state matrix in turn. The first row remains unchanged, but the first, second and third rows are shift 1, 2 and 3 positions to the left respectively.

$$M = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad \texttt{ShiftRows}(M) = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix} \tag{B.7}$$

B.2.4 `MixColumns` Step

`MixColumns` is a permutation that operates column by column of the state matrix. The `MixColumns` step takes four input bytes and produces four output bytes, whereby each input byte affects all of the output bytes. Each column of the state matrix (represented above by vector a) is treated as a polynomial over GF(2^8) and is multiplied modulo x^4+1 with a polynomial $p(x)$. The polynomial coefficients have simple values: 0, 1, 2 and 3. Coefficients of 0 and 1 result in no processing, a coefficient of 2 results in a shift to the left and a coefficient of 3 results in a shift to the left and XORing with the unshifted value. Let $b(x) = p(x) \times a(x) \pmod{x^4+1}$, then:

$$\begin{bmatrix} b_{0,0} \\ b_{0,1} \\ b_{0,2} \\ b_{0,3} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} a_{0,0} \\ a_{0,1} \\ a_{0,2} \\ a_{0,3} \end{bmatrix} \tag{B.8}$$

B.2.5 `AddRoundKey` Step

`AddRoundKey` is a simple XOR of the current round, a, with the subkey, k:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \oplus \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \tag{B.9}$$

B.3 Block Cipher Modes

A block cipher is a simple cryptographic primitive that can covert a fixed-length block of plaintext to a block of ciphertext. *Modes of operation* are used to specify how the cipher encrypts and decrypts blocks. This is especially important for ensuring the confidentiality of long messages [DR02, p. 27].

Using cipher block modes can help to prevent against some attacks such as frequency analysis. Block size can also be an important consideration, as small block sizes can be

vulnerable to attacks based on statistical analysis. Most block ciphers use a typical block size of 64-bits, however AES uses a larger 128-bits [MOVR96, p. 225].

The following are popular block cipher modes:

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Cipher Feed back mode (CFB)
- Output Feed Back mode (OFB)

The remainder of this section will explain how each of the above block modes worked. These explanations are based on [Sch96] and [MOVR96].

B.3.1 Electronic Code Book (ECB)

Electronic Code Book is by far the simplest method of encryption with a block cipher. With ECB each block is encrypted independently, whilst this can present some advantages, it is a method that is heavily undermined by the disadvantages.

The main advantage of ECB is that each block is independent, thus meaning that plaintext does not have to be encrypted or decrypted linearly (from start to finish). In some contexts the ability to randomly and independently access blocks of encrypted data can be useful but this does open up a number of vulnerabilities.

The independent nature of ECB means that two identical plaintext blocks will have identical ciphertext blocks - this is a significant problem. Patterns in plaintext data will still highly correlated between the plaintext and ciphertext blocks. If an attacker has access to several plaintext and ciphertext messages they can start to compile a codebook of known plaintext-ciphertext pairs.

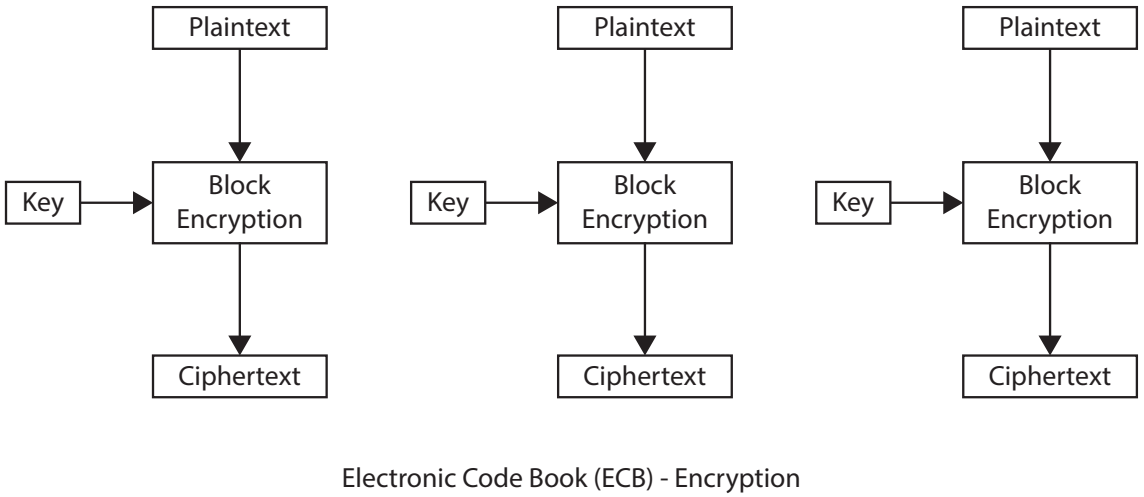


Figure B.1: Electronic Code Book (ECB) - Encryption

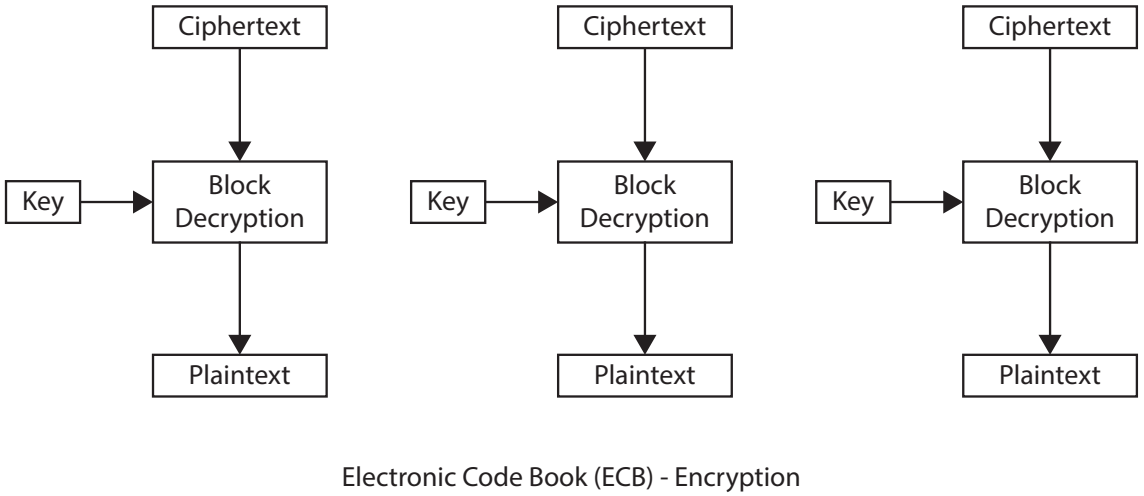


Figure B.2: Electronic Code Book (ECB) - Decryption

B.3.2 Cipher Block Chaining (CBC)

Invented by IBM in 1976, CBC ensures that each plaintext block is randomised before it is encrypted with the block cipher. CBC XORs each block of plaintext with the previous ciphertext block before encrypting with the block cipher. To ensure randomisation an Initialisation Vector (IV) is used for the first block. The CBC method makes each block of ciphertext dependent on all of the plaintext blocks up to that point. Unlike ECB, CBC must encrypt and decrypt linearly because of this dependancy.

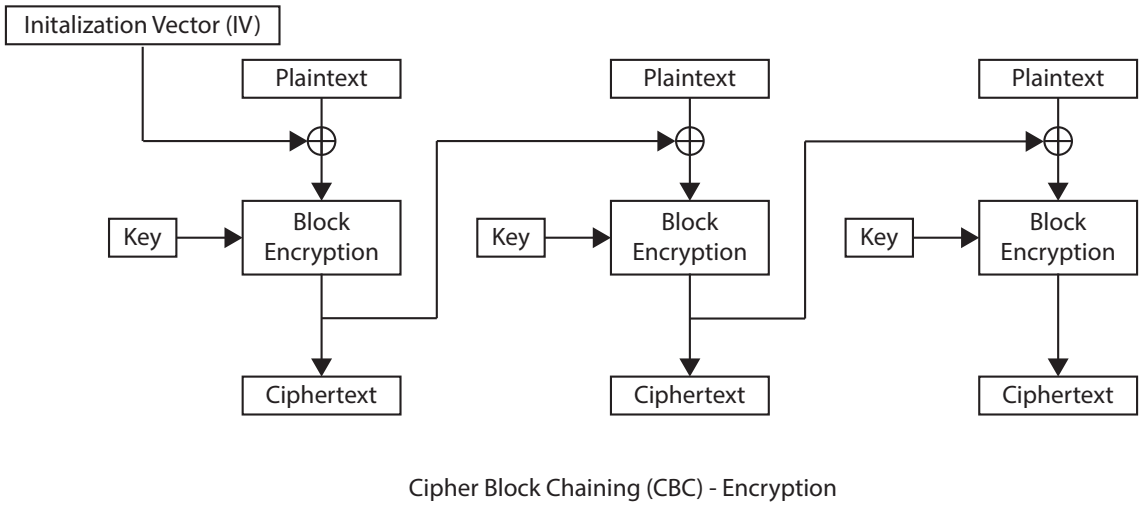


Figure B.3: Cipher Block Chaining (CBC) - Encryption

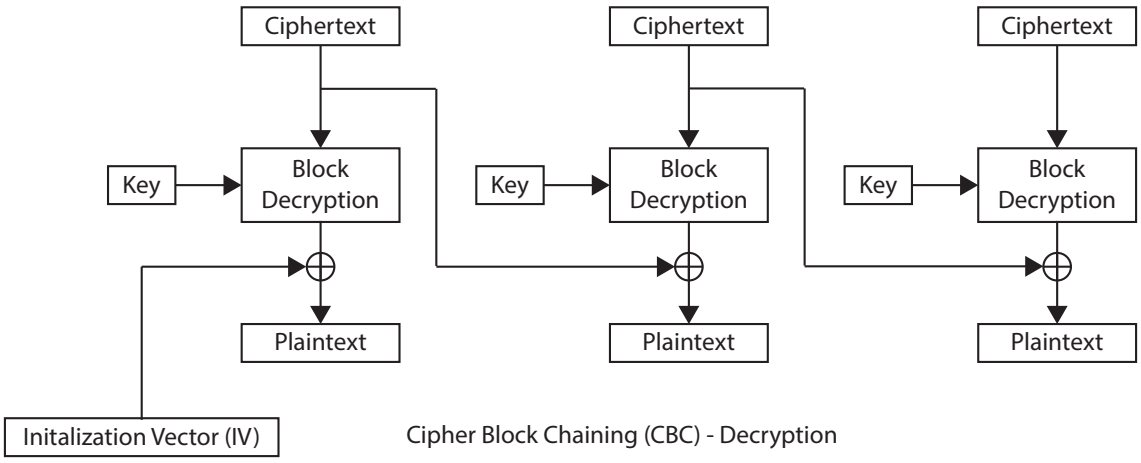


Figure B.4: Cipher Block Chaining (CBC) - Decryption

B.3.3 Cipher Feed Back mode (CFB)

ECB and CBC are modes specifically for block ciphers, but block ciphers can also use stream cipher modes. CFB is a *self-synchronising stream cipher* - using this mode will turn a block cipher into a self-synchronising stream cipher. Self-synchronising stream ciphers use previous ciphertext bits to generate the keystream bits.

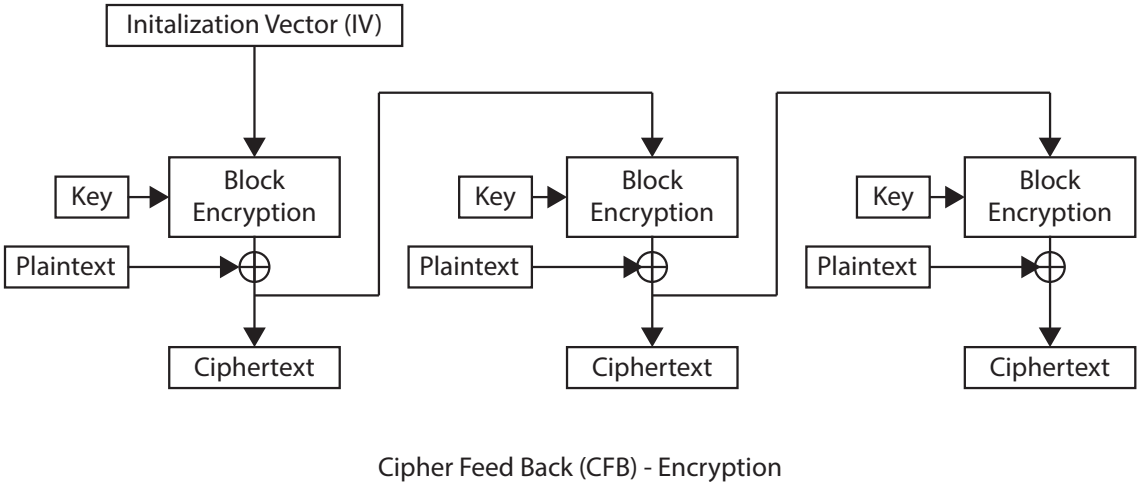


Figure B.5: Cipher Feed Back (CFB) - Encryption

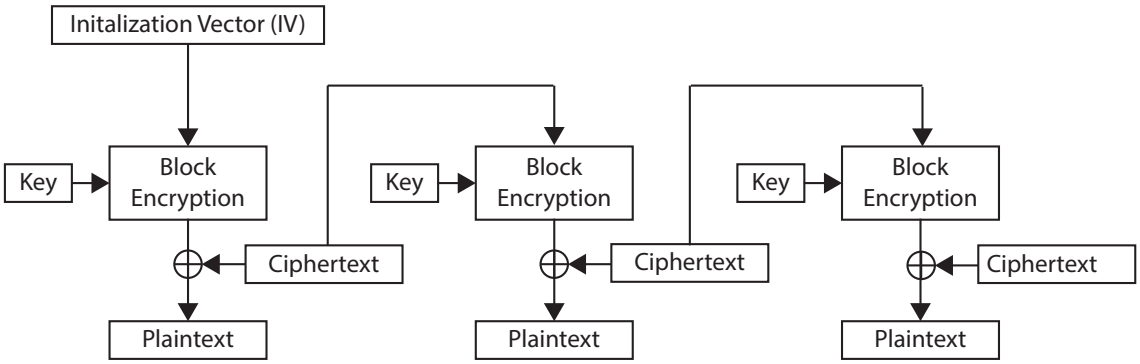


Figure B.6: Cipher Feed Back (CFB) - Decryption

B.3.4 Output Feed Back mode (OFB)

OFB is a synchronous stream cipher mode whose keystream is independent of the message being encrypted (independent of plaintext and ciphertext). OFB produces ciphertext by XORing the generated keystream with the plaintext.

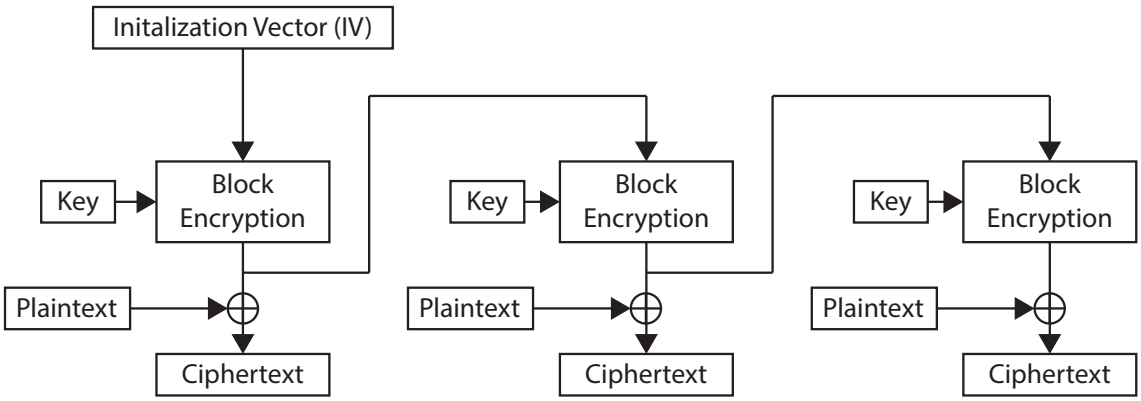


Figure B.7: Output Feed Back (OFB) - Encryption

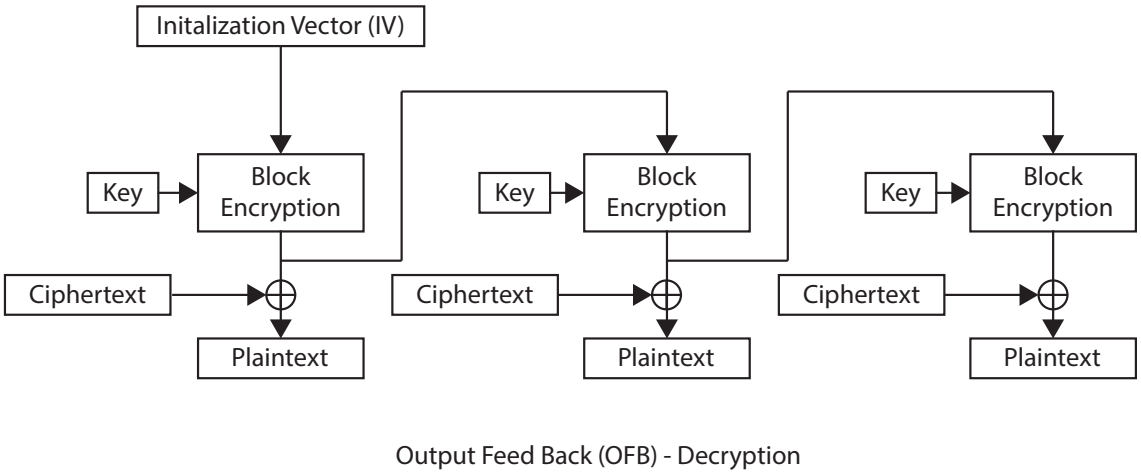


Figure B.8: Output Feed Back (OFB) - Decryption