Scott Williams

STEGANOGRAPHIC FILE SYSTEMS WITHIN VIDEO FILES

Computer Science Part II Project Dissertation

Christ's College
University of Cambridge

April 9, 2015

[Third Draft]

Proforma

NAME:	Scott Williams
COLLEGE:	Christ's
PROJECT TITLE:	Steganographic file systems within video files
EXAMINATION:	Part II of the Computer Science Tripos
YEAR:	2015
WORD COUNT:	13,758 (cut out 870)
PROJECT ORIGINATOR:	Scott Williams
PROJECT SUPERVISOR:	Daniel Thomas

Original Aims of the Project

To investigate appropriate steganographic embedding methods for video and to develop a practical steganographic software package to enable the embedding of arbitrary data within video files via a file system interface. Raw `AVI` video files should be supported and a variety of steganograhpic embedding algorithms should be available. Basic file system commands should work within the presented logical volume and embedding should occur with no perceivable impact on video quality. If time permits, multiple video formats should be supported along with encryption and plausible deniability functionality.

Work Completed

A complete software package has been developed enabling the embedding of arbitrary files within multiple video formats via a file system interface. A native uncompressed `AVI` decoder has been implemented along with a generic method for dealing with other video formats. A total of 6 steganographic embedding algorithms are supported, along with encryption and plausible deniability functionality. Common file system operations work as expected within the mounted volume and the embedding process can operate without any perceivable impact on video quality as verified by a user study. The performance of the file system is adequate for general use achieving read and write speeds on par with some USB 3.0 flash drives.

Special Difficulties

None.

Declaration of Originality

I, Scott Williams of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I give permission for my dissertation to be made available in the archive area of the Laboratory's website.

Signed:

Date:

Contents

1 || Introduction

Steganography is the art of hiding information in apparently inconspicuous objects. Whereas cryptography seeks to protect the content of information, steganography attempts to conceal the fact that the information exists[1, pp. xv - xvi]. This allows steganographic methods to be utilised in countries where encryption is illegal and identification of encrypted data can be grounds for imprisonment. Within the UK for example, keys for identified encrypted data can be forced to be disclosed[2], rendering standard cryptographic methods alone unfavourable.

In this project I design and implement a practical steganographic software application - `Stegasis` - which enables users to embed arbitrary files within video via a file system interface. `Stegasis` can operate with no perceivable impact on video quality and can provide very large embedding capacities. Multiple video formats are supported along with several steganographic embedding algorithms. Standard encryption algorithms can be used to further protect embedded data and plausible deniability functionality keeps sensitive information safe even when the presence of embedded data has been confirmed.

Steganogaphic methods operating on video have had comparatively little attention compared to other medium such as images and audio[1, p. xvii]. As such, there are few programs currently available which allow data to be steganographically hidden within video[3]. `Stegasis` is the first application to enable the embedding of arbitrary files within videos via a file system interface.

1.1 Motivation

Digital media is ubiquitous on the Internet and high definition video content is now common place on video sharing and social networking websites. Video files of multiple gigabytes in size can reside on users devices without arousing suspicion, providing an ideal hiding place for large collections of sensitive files. Few programs are capitalising on this fact, and those that are, allow the user to embed only a singe chosen file into a small range of video formats with very low embedding capacities. As with `TrueCrypt`[1], I believe that a practical system for protecting sensitive files should present the user with a mounted logical volume allowing the use of standard file system operations to create, access and organise embedded data. Furthermore, there exist many commonly used video formats along with many more currently in development. As such, a steganographic program operating on a small number of video formats not only greatly restricts usability, it will require constant development as new video formats inevitably become more popular. Instead, a generic solution applicable to a variety of video formats is preferred.

The recent global surveillance disclosures show the extent to which government authorities monitor online communications[5]. These, together with current UK laws mean it is no longer the case that simply encrypting data is enough to keep the owner safe.

[1]A successful widely used cryptographic program providing on-the-fly and full disc encryption[4]. Unfortunately, `TrueCrypt` is not longer being maintained.

2 || Preparation

2.1 Background

The project aims to combine 3 main topics; steganography, video formats and file systems. Understanding of these will be required to develop a successful final product.

The most important property of any steganographic system is undetectability, that is, it should be impossible to differentiate between ordinary and steganographically modified objects. This requirement is famously formulated within Simmons' prisoners' problem[6]:

Alice and Bob are imprisoned in separate cells and wish to formulate an escape plan. They are allowed to communicate, but all messages must pass through a warden Eve. If Eve suspects the prisoners of secretly discussing their escape plan, the communication channel will be severed and Alice and Bob thrown into solitary confinement. The prisoners attempt to utilise steganograhpy to exchange details of their plan undetected. The steganographic system is considered broken if Eve is able to detect the presence of hidden messages within the prisoners exchanges. It is assumed that Eve has a complete knowledge of the steganographic algorithm being used, with the exception of the stego key, which Alice and Bob have agreed upon beforehand. This is in parallel with Kerckhoff's principle used within cryptography[7]. The warden can be considered to be one of three categories: *passive*, *active* and *malicious*. A passive warden does not modify the exchanged messages in any way, whereas an active warden may modify the messages whilst maintaining their original meaning. For example replacing words with synonyms or reordering sentences. If images are being used as a transport medium then an active warden may recompress or crop the images. A malicious warden attempts to break the steganograhpic system and impersonate the prisoners in an attempt to obtain information.

This project is concerned only with the case of the passive warden. As such, any modification of the video files once `Stegasis` has embedded data within them, will most likely render the embedded file system corrupt. This unfortunately means utilising video sharing websites such as YouTube and Facebook for distribution is not possible due to them performing compression and transcoding upon video upload.

2.1.1 Steganographic Concepts

A *steganographic system* will form a core part of the final application, allowing requested data to be hidden within video. A steganographic system depends on the following components:

- A *Cover object* is the original object that the message will be embedded within. A cover object consists of a number of *elements*, for example pixels.
- A *Message* is an arbitrary length sequence of symbols. For this project we consider messages of the form $\mathcal{M} \in \{0,1\}^{8 \cdot n}$ for some n - a sequence of bytes.
- A *Stego key* is a secret key used within the embedding process.

- A *Stego object* is the result of embedding a message inside a cover object.

Definition 2.1. Steganographic System

Let $\mathcal{C}$ be the set of all cover objects. For a given $\boldsymbol{c} \in \mathcal{C}$, let $\mathcal{K}_{\boldsymbol{c}}$ denote the set of all stego keys for $\boldsymbol{c}$, and the set $\mathcal{M}_{\boldsymbol{c}}$ denote all messages that can be communicated in $\boldsymbol{c}$. A steganographic system[2] is then formally defined as a pair of embedding and extracting functions Emb and Ext,

$$\begin{gathered} Emb : \mathcal{C} \times \mathcal{K} \times \mathcal{M} \to \mathcal{C} \\ Ext : \mathcal{C} \times \mathcal{K} \to \mathcal{M} \end{gathered}$$

satisfying,

$$\forall\ \boldsymbol{c} \in \mathcal{C}, \boldsymbol{k} \in \mathcal{K}_{\boldsymbol{c}}, \boldsymbol{m} \in \mathcal{M}_{\boldsymbol{c}}.\ Ext(Emb(\boldsymbol{c},\ \boldsymbol{k},\ \boldsymbol{m}),\ \boldsymbol{k}) = \boldsymbol{m}$$

Definition 2.2. Embedding Capacity

The Embedding Capacity (payload) $\mathcal{P}_{\boldsymbol{c}}$ for a given cover object $\mathbf{c} \in \mathcal{C}$ is defined in bits as,

$$\mathcal{P}_{\boldsymbol{c}} = \log_2 |\mathcal{M}_{\boldsymbol{c}}|$$

The relative embedding capacity $\mathcal{R}_{\boldsymbol{c}}$ for a given cover object $\mathbf{c} \in \mathcal{C}$ is defined as,

$$\mathcal{R}_{\boldsymbol{c}} = \frac{\log_2 |\mathcal{M}_{\boldsymbol{c}}|}{n}$$

where n is the number of elements in $\mathbf{c}$.

For example, consider $\mathcal{C}$ to be the set of all 512×512 greyscale images, embedding one bit per pixel gives $\mathcal{M} = \{0,1\}^{512\times512}$ and $\forall\ \mathbf{c} \in \mathcal{C}.\ |\mathcal{M}(\mathbf{c})| = 2^{512\times512}$. The embedding capacity $\forall\ \mathbf{c} \in \mathcal{C}$ is then $512 \times 512 \approx 33$kB as expected. In this case, n is equal to the number of pixels in $\mathbf{c}$ and therefore the relative embedding capacity is equal to 1 bpp (bits per pixel), again as expected.

Using the definitions above, we can define a simple expression for the embedding capacity of a video file.

Definition 2.3. Embedding Capacity for video

With $\mathcal{C}$ as the set of all video files, the embedding capacity $\mathcal{V}_{\boldsymbol{c}}$ for a given video $\mathbf{c} \in \mathcal{C}$ can be expressed as,

$$\mathcal{V}_{\boldsymbol{c}} \ = \sum_{f\ \in\ \text{frames}(\boldsymbol{c})} \mathcal{P}_f$$

Note that for certain embedding algorithms, the embedding capacity can depend on both the input data and the cover object[3]. However, in some cases the following expression is also valid,

$$\mathcal{V}_{\mathbf{c}} \ = \ |\text{frames}(\mathbf{c})| \cdot \mathcal{P}_{f_0}$$

[2] This is specifically steganography by cover modification.

[3] Many algorithms operating on `JPEG` images for example will not embed within zero valued DCT coefficients.

Definition 2.4. Steganographic Capacity
The concept of Steganographic Capacity is loosely defined as the maximum number of bits that can be embedded within a given cover object without introducing statistically detectable artifacts.

For completeness, the least significant bit (LSB) of a given number is defined as follows,

$$\mathrm{LSB}(x) = x \mod 2$$

It will be useful to visually inspect the effect of steganographic embedding algorithms operating on the LSBs of pixels. The *LSB Plane* of an image is therefore defined.

Definition 2.5. LSB Plane
The Least Significant Bit Plane of a given image $\boldsymbol{c}$ *and a specified colour channel* q *is defined as the 1 bit image* $\mathrm{LSBP}(\boldsymbol{c}, q)$ *which has resolution equal to that of image* $\boldsymbol{c}$ *and with pixel values* $\mathrm{LSBP}(\boldsymbol{c}, q)(x, y)$ *given by,*

$$\mathrm{LSBP}(\boldsymbol{c}, q)(x, y) = \mathrm{LSB}(\boldsymbol{c}(x, y))$$

2.1.2 Steganalysis

Steganalysis is the study of detecting messages embedded using steganographic techniques; this is analogous to cryptanalysis applied to cryptography[8]. A steganalysis attack is considered successful (that is, the steganography has been broken) if it is possible to correctly distinguish between cover and stego objects with probability better than random guessing. Note that it is not necessary to be able to read the contents of the secret message to break a steganographic system.

A trivial example of steganalysis arises when the steganalyst has access to the original cover object used within the embedding procedure. By computing the difference between the stego and cover objects, the steganalyst can immediately detect the presence of a hidden message. This attack identifies a number of important points to consider when developing a practical steganographic system. Firstly, embedding within popular media content should be discouraged, as the cover object will likely be widely available. Secondly, if a user is embedding within original content, for example a video recorded by them, any copies of the original file should be securely erased after embedding.

Steganalysis methods can be split into two main categories, *Targeted Steganalysis* and *Blind Steganalysis*. Targeted Steganalysis occurs when the steganalyst has access to the details of the steganographic algorithm used for embedding. The steganalyst can accordingly target their activity to the specific stegosystem. On the other hand, if the steganalyst has no knowledge of the utilised steganographic algorithm, Blind Steganalysis techniques must be applied. In this project, Targeted Steganalysis attacks are developed for several of the proposed embedding algorithms.

2.1.3 The AVI file format

As specified within the project proposal, this project initially looks at raw uncompressed `AVI` files. Furthermore, only `AVI` version 1.0[4] files are investigated and therefore supported natively[5] by `Stegasis`. Unfortunately, uncompressed `AVI` is today, a very uncommon video format[9]. This is likely due to its relatively huge file sizes when compared to a modern compressed format such as `H.264`. For example, one minute of 720p HD footage encoded as uncompressed `AVI` is roughly 4.2 GB.

The `AVI` file format is a Resource Interchange File Format (`RIFF`) file specification developed by Microsoft and originally introduced in November 1992[10]. The data within `RIFF` files is divided into chunks and lists, each of which is identified by a `FourCC` tag. An `AVI` file takes the form of a single chunk in a `RIFF` formatted file, which is then subdivided into two mandatory lists, the `hdrl` and `movi` and one optional chunk, the `idxl`. The second sub-list contains the actual audio/video data and will be where steganographic embedding will occur. See the Appendix Section A for detailed definitions of the data structures used.

An `AVI` file consists of a number of data streams (usually 2, one for audio and one for video) interleaved within the `movi` list. Each stream has a corresponding `AVI` stream header and format chunk within the above mentioned `hdrl` list. These data structures contain information about the stream including the codec and compression used (if any). Specifically, the `fccHandler` field contains a `FourCC` tag that identifies a specific data handler. For raw uncompressed video this will equal 'DIB ' (Device Independent Bitmap). Any user provided `AVI` files with a `fccHandler` not equal to 'DIB ', (compressed video) will at this point be rejected and an error message presented to the user.

The `movi` list contains the raw video and audio data within sequential `RIFF` chunks. Each chunk for the DIB video stream contains one frames worth of pixel data, with each pixel represented by a 3 byte BGR (Blue Green Red) triple - a total of 24 bits per pixel. The first 3 byte triple corresponds to the lower left pixel of the final image[6].

If we use an embedding algorithm which embeds 3 bits per pixel (1 bit per colour channel per pixel), we can derive a simple expression for the embedding capacity of an uncompressed `AVI` video $\mathbf{c}$, in terms of the height h and width w in pixels and the total number of frames t:

$$\mathcal{V}_{\mathbf{c}} = 3 \cdot w \cdot h \cdot t$$

These values are all available within the `AVIMAINHEADER` structure allowing the user to be informed of the video's total embedding capacity upon formatting.

[4]Not including the Open-DML extension (version 1.02).
[5]All other video formats (including compressed `AVI`) are supported via the use of `FFmpeg`, as described in Section 3.6.1.
[6]This can be inverted via a flag within the `BITMAPINFOHEADER`.

2.1.4 JPEG compression

The JPEG file format will prove useful when developing a universal steganographic technique operating across many video formats (Section 3.6.1). Steganography within the JPEG format has had a comparatively large amount of attention from the research community[11]. As such, there exists a fair number of well documented steganographic embedding algorithms for JPEG[12] [14] [13].

The JPEG compression process consists of 5 main procedures:

1. Transform the image into an optimal color space.
2. Downsample chrominance components by averaging groups of pixels together.
3. Apply a *Discrete Cosine Transform* (DCT) to blocks of pixels.
4. Quantise each block of DCT coefficients using a quantisation table.
5. Encode the resulting coefficients using a Huffman variable word-length algorithm.

Note that step 4 is an example of lossy compression, whereas step 5 is lossless. Therefore most steganographic algorithms will operate on the quantised DCT coefficients (between steps 4 and 5) to avoid embedded data being lost due to quantisation.

Conveniently, the Independent JPEG Group provide the `libjpeg` C library[15] which will abstract the complexities of the JPEG format and allow direct access to the quantised DCT coefficients prior to step 5 being executed.

JPEG DCT coefficients are arranged into several components containing *rows* which contain a number of *blocks*. Each block contains 64 coefficients ranging from -2048 to 2047. A JPEG will usually have 3 components corresponding to the *Luminance and Chrominance* colour model (`YCbCr`) with the first component being luma and the remaining two being the blue-difference and red-difference chroma components. Since human perception is more sensitive to changes in luminance compared to colour[16], steganographic embedding will usually not occur within the luminance component.

It is worth noting that the JPEG decompression and compression processes are computationally expensive. This is especially important when dealing with video since a 3 minute music video, for example, consists of around 4,500 frames (which can be considered as individual JPEGs). Since performance of the virtual file system is important, design decisions will need to be made to avoid any unnecessary compression and decompression operations. Also worth noting is that although JPEG files are small on disk, they are not once decompressed into memory. It will not be possible to hold all 4,500 decompressed JPEG frames of an average 3 minute video in memory.

2.1.5 FFmpeg

`FFmpeg` is an open source, multimedia framework[17]. It is a “complete, cross-platform solution to record, convert and stream audio and video”. In particular, it contains codecs for nearly every video format available today[18].

The pitfalls of the uncompressed AVI video format, as discussed in Section 2.1.3, show that Stegasis would greatly benefit from operating on multiple video formats. I could continue to investigate more video formats and develop parsers for these as part of the project. However, this will become a very time consuming endeavor most likely resulting in very brittle, untested parsers. Instead, it would be wise to leverage the FFmpeg framework for this functionality.

One trivial solution to allow Stegasis to operate on multiple video formats would be to convert all user provided video files to uncompressed AVI prior to the embedding process. However, this doesn't solve the problems of the huge file sizes and uncommonality of the uncompressed AVI format.

A novel solution to this problem is posed in Section 3.6.1 and makes use of FFmpeg to convert input files to the *Motion JPEG* file format.

2.1.6 Developing a file system

A file system can either operate within *kernel* or *user space*. I decided within the project proposal to develop the file system component for Stegasis in user space using the FUSE (Filesystem in Userspace) library[19]. This was primarily because developing a kernel module is complex and hard to test - a segmentation fault occurring within kernel space code will bring down the entire machine. A kernel module also requires a large amount of boiler plate code and I would prefer to spend time on the steganographic portion of this project rather than getting bogged down with the complexities of a kernel file system implementation. In contrast, FUSE ships with an example "hello world" file system which is less than 100 lines of C code. Also, developing the file system in user space will cause the final application to be a lot more portable and easier for users to install - a kernel space file system would require super user permission to load the related kernel module.

There are however disadvantages to using a file system in user space, performance being one of them[20]. This is due to the FUSE kernel module having to act as a proxy between the system call and the user space code.

Figure 2.1 shows the path of a file system call in the provided hello world example file system. We can see the FUSE kernel module acting as a proxy between the VFS system call and the example/hello user space code. A kernel space file system would not need to re-enter user space to complete the system call, hence giving better performance.

The FUSE library provides a number of function definitions which the user space code implements. These functions are then called when the corresponding file system operation occurs.

```c
int read(const char *path, char *buf, size_t size, off_t offset, struct
    fuse_file_info *fi);
```

Listing 2.1: FUSE read operation.

The read function is called when a file system read occurs. It requests that size bytes of the file path starting at offset offset should be written to the buffer buf.

images/fuse_structure.png

Figure 2.1: Path of a file system call in FUSE, taken from fuse.sourceforge.net.

2.2 Existing tools

The relatively little work on steganography within video was reflected in my search for steganographic programs operating on video files. This section contains an exhaustive list of all the video steganogrphy tools I could find freely[7] available on the Internet. A total of 6 tools claimed to provide steganographic embedding functionality within video files. Of these 6, only 3 actually attempt to embed within the video data itself. None of the identified programs allow the user to embed more than one file[8] and none of them provide any sort of file system interface.

2.2.1 StegoStick

StegoStick[21] claims to allow users to "hide any file into any file". This statement suggests that the program is simply appending the requested file to the end of the cover object. This suspicion is partly true; based on the file extension, StegoStick splits cover objects into 3 categories: images, media and other. The other category does indeed just append the file to the cover object, whereas the image and media category do attempt to employ steganographic embedding methods. The images category applies to files with

[7]A further 2 programs exist claiming to embed within video, however these are closed source and not freely available to download. Therefore they have been excluded from this list. (Info Stego, Hiderman)

[8]Admittedly you could embed a compressed archive using these tools to effectively allow a directory structure to be embedded.

extensions JPG, GIF and BMP and uses LSB embedding within BMP files (other image formats are converted to BMP prior to embedding). The media category applies to WAV, AVI and MPG files and assumes each format has a “header” of 44+55 bytes[9]. Although this seems to be true for the WAV format, this is not the case for AVI nor MPG files. StegoStick will then use blind LSB embedding within the remaining data. As such, my attempts to use StegoStick to embed within AVI files rendered the resulting video unplayable.

2.2.2 StegoMagic

StegoMagic[22] claims to “work on all types of files and all size of data” which again sounds as though it’s appending the file to the end of the cover object. This is indeed the case, embedding an image within a video and inspecting the modified file shows that data has just been appended to the end of the video, albeit encrypted. StegoMagic does not specify the encryption algorithm used and the source code is not available to view. Furthermore, the user cannot specify an encryption key to use. Instead, StegoMagic generates a 5 digit number during the embedding process and presents this to the user.

2.2.3 TCSteg

TCSteg[23] is a Python script accompanying a blog post written by Martin Fiedler discussing hiding TrueCrypt volumes within MP4 files. The method described embeds the TrueCrypt volume within the MP4 atom mdat and modifies the chunk offset table within the moov atom so that any application playing the video will ignore the embedded data. A nice property of TCSteg is that the resulting video file can be directly mounted by TrueCrypt since it ignores the MP4 header data prior to the embedded volume.

The above programs all resort to embedding within video files by either appending the embedded data to the end of the video, or inserting the embedded data at some point within the video file. I do not consider this approach to embedding data secure, and it should be a trivial task for any steganalyst to detect the presence of embedded data within the stego objects using a simple hex editor. Therefore, the above stegosystems should be considered broken and definitely not used for the hiding of sensitive data.

2.2.4 StegoVideo

StegoVideo[24] is a Virtual Dub filter[10] which allows users to embedded a file within AVI files (supporting multiple compression codecs). I am unsure of the exact steganograhpic embedding algorithm used since the program is closed source, but the website does mention that StegoVideo makes use of error correction codes to allow embedded data to be recovered even after the resulting video has been compressed - although this is understandably dependant on the compression amount. StegoVideo attempts to protect

[9] Listed in the source as “44 byte header + 54 bytes of extension space”.

[10] Which is also available in a stand alone executable form.

the embedded data via the use of a 5 digit number, although as with `StegoMagic`, this is not provided by the user and is instead generated and presented to the user.

2.2.5 OpenPuff

`OpenPuff`[25] is a steganographic tool supporting a wide range of formats, including `3GP`, `MP4`, `MPG` and `VOB`. It allows users to embed a file within a collection of carrier objects and uses 3 user provided passwords to encrypt, scramble and whiten (mixing with a high amount of noise) the provided file. Plausible deniability is also provided via the option to add decoy content. `OpenPuff` successfully embedded and retrieved a text file within a sample `MP4` video and I could notice no perceivable impact on video quality. Performance was also good due to multithreading support. However, the embedding capacity is very limited. A hard limit of 256 MB is imposed regardless of the number and size of the carrier objects and I was only able to achieve embedding capacities of around 0.0043%[11] even at the maximum capacity setting. This makes `OpenPuff` impractical for hiding large files - for example, you would need around 770 60 MB `MP4` carrier files to embed a standard 2 MB `JPEG` image.

2.2.6 Steganosaurus

`Steganosaurus`[26] is a cross platform steganograhpic program developed by James Ridgway. It allows users to embed a file within `H264` video files via the modification of motion vectors. Two embedding algorithm variants are provided and the input file is encrypted using `AES` with a user provided passphrase. A modified version of `FFmpeg` was used to access and modify the motion vectors, these modifications have not yet been open sourced. I unfortunately could not get `Steganosaurus` to run on my computer (using Linux or Windows) and therefore could not test its operation.

The above 3 programs are much more promising from a steganogrphic security point of view and some of them also support multiple video formats. However, all feature the same limitation of only allowing the user to embed one chosen file and the offered embedding capacities are far from practical for use with large files.

This project aims to remedy these issues by allowing the user to embed an arbitrary number of files within a video (via a file system interface) and by providing high capacity steganographic embedding algorithms (for example 15% of the video size)[12].

2.3 Requirements Analysis

After reviewing the necessary background material and investigating current available solutions to the problem of steganography within video, I produced the following collection

[11] 2,600 bytes within a 60 MB video.

[12] This is very much a trade off - larger embedding capacities will come at the sacrifice of steganogrphic security. However, this decision is presented to the user rather than decided by the program itself and sensible defaults will be put in place.

of requirements. For the project to be a success, all of the core requirements should be fulfilled.

2.3.1 Core Requirements

`Stegasis` should:

1. Allow users to embed data within video files:
    a) Several steganographic embedding algorithms should be available.
    b) Each embedding algorithm, $\mathcal{A}$, should satisfy correctness. That is,
    $$\forall\ \mathbf{c}, \mathbf{k}, \mathbf{m}.\ Ext_{\mathcal{A}}(Emb_{\mathcal{A}}(\mathbf{c}, \mathbf{k}, \mathbf{m}), \mathbf{k}) = \mathbf{m}.$$
    c) Embedding should occur with no perceivable impact on video quality.
    d) Steganalysis tools should be developed to test the security of the proposed embedding algorithms.
    e) An optional user provided password should encrypt data prior to embedding.
    f) A capacity flag should allow users to specify the percentage of each video frame to embed within.
2. Provide a file system interface:
    a) The presented logical volume should reside at a user provided mount point.
    b) Data written to the file system should be embedded on the fly within the chosen video file.
    c) Data accessed from the file system should be extracted on the fly from within the video.
    d) Standard file system operations such as creating, deleting and moving files should work as expected, and standard Unix tools such as `cp`, `mv` and `rm` should also work as expected.
3. Support raw uncompressed `AVI` video:
    a) Uncompressed `AVIs` should be natively decoded allowing access to individual pixel data.
4. Provide adequate file system performance:
    a) Full HD video content should be playable directly from within the presented file system (bitrates of full HD video are roughly 8 - 12 Mb/s[27]).
    b) Ideally, the file system should provide read and write speeds comparable to those provided by USB 2.0 devices[13](roughly 20 MB/s[28]).

[13] Although the USB 2.0 standard supports speeds of up to 480 Mb/s, devices rarely reach this theoretical limit.

2.3.2 Possible Extensions

If time constraints allow, the following extension tasks shall also be completed.

`Stegasis` should:

1. Support embedding within multiple video formats.
2. Allow directory operations within the file system:
   a) Creating directories using the `mkdir` command should work as expected, as should using the `mv` and `rm` commands.
   b) Organising files within directories should also work as expected.
3. Embed also within audio data:
   a) Data should also be embedded within the (possible) audio stream of the video, therefore increasing the embedding capacity.
4. Provide plausible deniability:
   a) A second file system should be (optionally) embedded within the video, mountable with a second passphrase.
   b) The presence of the second, hidden file system should not be detectable.
5. Be evaluated for perceivable video impact using an evaluation study:
   a) A developed web application should evaluate the requirement "Embedding should occur with no perceivable impact on video quality." by obtaining data from multiple users.

2.4 Choice of Languages and Tools

With the above requirements for the final product defined, an appropriate set of programming languages and tools can be identified.

`Stegasis` will be designed to operate on the `Linux` operating system since `Windows` has no equivalent of the file system in user space paradigm.

Several of the core (and extension) requirements strongly suggest a lower level language such as C or C++ rather than a higher level sandboxed language such as Java. For example, the parsing and modification of `AVI` files lends itself to a language like C since it will involve large amounts of byte level manipulation. Furthermore, the Microsoft file format reference defines the different data structures used within `AVIs` as C `structs`. `FUSE`, `libjpeg` and libraries provided by `FFmpeg` are all natively C libraries. Although wrappers for other languages (including Java) do exist[29] [30], they seem to be lacking documentation and few are being actively maintained. The requirement that `Stegasis` should support several steganographic embedding algorithms implores the use of object oriented techniques; defining a `Steganographic Algorithm` interface of which each embedding algorithm implements. This suggests C++ over C. The final core requirement, performance, also favours C/C++ over Java[14] due to the JVM overheads.

The reasons above and the fact I have prior experience using C++ led to the conclusion that C++ should be the primary language used to develop `Stegasis`.

As discussed in Section 2.1.5, `FFmpeg` will be used for the extension task "`Stegasis` should support a wide range of video formats", to allow the decoding and conversion of the many video formats available today, together with library `libjpeg` discussed in Section 2.1.4 for the manipulation of `JPEG` images.

During the implementation of `Stegasis`, a number of small steganalysis programs will be developed. These will likely be written in a scripting language such as Python or Matlab since both have extensive library support for mathematical operations.

The extension task "`Stegasis` should be evaluated for perceivable video impact using a web application" will require a web application to be developed and hosted for easy access to participants and a database to store the collected user data. `Node.js` together with the web application framework `Express` and the database `MongoDB` was chosen as the development stack for the site. This decision was mainly due to the speed at which you can develop *CRUD* (create, read, update and delete) web applications - essentially what this evaluation site is - and my previous experience with the technologies.

[14] Numerous studies have shown that C/C++ code performs better than equivalent Java code[31] [32] [33].

3 || Implementation

3.1 Introduction

The development of `Stegasis` consisted of the five main stages detailed within this chapter. Firstly, a parser for the `AVI` file format as discussed in Section 2.1.3 was developed allowing direct access to video pixel data. Next, steganographic embedding algorithms were implemented along with corresponding steganalysis tools to test the security of the proposed techniques. The file system was then developed utilising the `AVI` decoder and steganogrphic algorithms to embed and extract data directly into and out of video files. Finally, the extension tasks were individually addressed providing support for multiple video formats, directory structures and plausible deniability.[15] The testing section provides an overview of the testing processes applied throughout development.

The software development process embraced the modern "Launch early, iterate often" methodology[34]. A simplified version of `Stegasis` was initially produced allowing integration issues to be identified early on, when the code was still very malleable. Once this basic version was working, an iterative approach was then taken to add more functionality and features.

3.2 AVI Decoder

The concept of an `AVI` decoder is first abstracted to that of a generic `Video Decoder` interface[16]. The core requirements state that the `AVI` decoder should allow access to individual pixel data. The pixel data within an `AVI` file is grouped into chunks, one per video frame. This lead to the decision to define the `Video Decoder` to allow access to the video pixel data at a granularity of a single video frame. It will also be useful for the `Video Decoder` interface to expose metadata about the video, for example, the total number of video frames in the video, the height and width of the video frames and the total size (in bytes) of each video frame.

This gives the definition for the `Video Decoder` interface as described in Listing 3.1 (`NextFrameOffset` will be discussed in Section 3.4).

Note that `getFrame` returns a `Frame` wrapper object, rather than a raw `char` pointer to the frame pixel data, adhering to the *Dependency Inversion* principle[35]. This will be useful when dealing with different video formats that don't necessarily group all of a frames video data to be accessible by a single `char` pointer.

A `Frame` abstracts the concept of a single frames video data. In the case of uncompressed `AVI`, this can be thought of as a `char` pointer to the GBR pixel data, along with an associated frame size in bytes. A boolean value is also associated with each chunk, signifying if the chunk data has been modified, that is, it is dirty. The `Frame` interface is defined in Listing 3.2.

[15] The evaluation site extension task is discussed within the evaluation chapter.

[16] The term "interface" is used as shorthand for an abstract C++ base class. That is, a class with pure virtual member functions and no function implementations.

```
class VideoDecoder {
  public:
    virtual Frame *getFrame(int frame) = 0;
    virtual int getFileSize() = 0;
    virtual int getNumberOfFrames() = 0;
    virtual int getFrameSize() = 0;
    virtual int getFrameHeight() = 0;
    virtual int getFrameWidth() = 0;

    virtual void getCapacity(char capacity) = 0;
    virtual void setCapacity(char capacity) = 0;

    virtual void getNextFrameOffset(int *frame, int *offset) = 0;
    virtual void setNextFrameOffset(int frame, int offset) = 0;

    virtual void writeBack() = 0;
    virtual ~VideoDecoder() {};
};
```

Listing 3.1: `Video Decoder` interface (`video/video_decoder.h:15`)

```
class Frame {
  protected:
    long frameSize;
  public:
    virtual long getFrameSize() = 0;
    virtual char *getFrameData(int n=0, int c=0) = 0;

    virtual bool isDirty() = 0;
    virtual void setDirty() = 0;
};
```

Listing 3.2: Frame interface (`video/video_decoder.h:4`)

Note that the parameters for `getFrameData` are optional. For the `AVI` decoder, these will not be used.

The `AVI` parsing process can be thought of consisting of two main parts; parsing the video headers and parsing the video chunk data. The pseudocode in Algorithm 3.1 illustrates this with the headers being parsed lines 1-12 and the chunks being parsed lines 15-22. See the Appendix Section C for some longer code samples.

The actual implementation is slightly more complex than presented above due to the existence of `JUNK` chunks. The `AVI` file format specifies that any number of chunks with a `FourCC` code of `JUNK` and of arbitrary length can be inserted between any `AVI` list structures. The parser must therefore be able to cope with this.

The `WriteBack` function of the `AVI` decoder will write back any modified `Frame` data into the original `AVI` file. This operation is described in Algorithm 3.2 below.

`getFrameSize` returns the number of bytes within each frame that should be embedded within and is shown in Listing 3.3.

Algorithm 3.1 `AVI` parsing process

```
f ← open(file_path)
riff_header ← readRiffHeader(f)
if riff_header.fourCC != RIFF then
    Error "File is not an AVI file"
avi_header ← readAviHeader(f)
bitmap_info_header ← readBitmapInfoHeader(f)
if bitmap_info_header.compression != 0 then
    Error "Stegasis does not natively support compressed AVI files"
audio_info_header ← readAudioInfoHeader(f)
frame_chunks ← [ ]
i ← 0          ▷ File pointer is now positioned at the start of the audio video chunks
while i < avi_header.total_frames do
    chunk ← readChunk(f)
    if chunk is a video chunk then
        frame_chunks[i].chunkSize = chunk.chunkSize
        frame_chunks[i].frameData = readChunkData(f)
        i ++
    else
        Advance f chunk.chunkSize bytes
```

Algorithm 3.2 `AVI` write back process

```
Seek f to the chunks offset
i ← 0
while i < avi_header.total_frames do
    chunk ← readChunk(f)
    if chunk.fourCC == 00db then
        if frame_chunks[i].isDirty then
            Write frame_chunks[i].frameData to f
            frame_chunks[i].dirty = false          ▷ This chunk is no longer dirty
        else
            Advance f chunk.chunkSize bytes        ▷ Chunk did not need to be written
        i ++
    else
        Advance f chunk.chunkSize bytes            ▷ Chunk was not a video chunk
```

```
virtual int frameSize() {
  return (int)floor(this->aviHeader.width * this->aviHeader.height * 3 * (
      this->capacity / 100.0));
};
```

Listing 3.3: `AVI` decoder `frameSize` function (`video/avi_decoder.cc:298`)

This expression arises from the fact that uncompressed `AVI` uses 24 bits per pixel value. Since there are *height* · *width* pixels within a single frame, multiplying this by 3 will give the total number of bytes. *Capacity* is a user provided percentage ranging in value from 1 - 100. It specifies the percentage of the frame to embed within. `frameSize` must therefore reduce the returned frame size value by "capacity percent".

The effect of the capacity parameter is illustrated within Figure 3.1. The top left image is the original video frame and the top right image is the LSB plane (red channel) of the frame with no data embedded. The bottom two images have data sequentially embedded within them using capacity settings of 50% and 15% respectively.

images/cap_ill.png

Figure 3.1: Illustration of the capacity parameter.

The image used throughout this chapter for illustration purposes has been specifically chosen to emphasise the effect of the embedding process. Due to post processing applied to the video, the LSB plane closely resembles that of the original frame. This would not be the case for a video recorded on a phone for example.

The `AVI` decoder as described now provides all necessary functionality to allow the modification of pixel data to achieve the steganogrphic embedding of information within video frames.

3.3 Steganographic Algorithms

As defined within the previous section, a steganogrphic system consists of a pair of functions providing embedding and extraction functionality. Therefore, a generic `Steganographic Algorithm` interface will need to declare two functions `embed` and extract which will embed and extract data into and out of cover objects. Listing 3.4 shows the interface.

```
class SteganographicAlgorithm {
  protected:
    string password;
    VideoDecoder *dec;
  public:
    virtual pair<int, int> embed(Frame *f, char *data, int reqByteCount,
        int offset) = 0;
    virtual pair<int,int> extract(Frame *f, char *output, int reqByteCount,
         int offset) = 0;
    virtual void getAlgorithmCode(char out[4]) = 0;
};
```

Listing 3.4: Stego Algorithm interface (steg/steganographic_algorithm.h:8)

The embed function attempts to embed reqByteCount bytes from data into frame f starting at an offset offset bytes into the frame and returns the number of bytes successfully embedded along with the reached frame offset. Similarly, the extract function attempts to extract reqByteCount bytes from frame f starting at an offset offset bytes into the frame and put them into output and returns the number of bytes successfully extracted along with the reached frame offset.

getAlgorithmCode returns a 4 character algorithm identifier which is used when users specify which algorithm they want to use.

3.3.1 LSB Sequential Embedding

Sequential LSB embedding is arguably the simplest steganographic algorithm. It works by replacing the LSBs of the cover object with the bits comprising the message, producing the stego image. Algorithm 3.3 shows pseudocode for the LSB embedding algorithm. The matching extraction algorithm is shown in Algorithm 3.4.

Algorithm 3.3 LSB embedding algorithm

```
for i ← 0 upto dataBytes - 1 do
    for j ← 7 downto 0 do
        if The jth significant bit of data[i] == 1 then
            Set LSB(frame[offset++]) to 1
        else
            Set LSB(frame[offset++]) to 0
```

Algorithm 3.4 LSB extraction algorithm

```
for i ← 0 upto dataBytes - 1 do
    for j ← 7 downto 0 do
        Set the jth significant bit of output[i] to LSB(frame[offset++])
```

Listing 3.5 shows the actual implementation and Figure 3.2 illustrates the algorithms operation.

```
virtual pair<int,int> embed(Frame *f, char *data, int reqByteCount, int
    offset) {
  this->crypt->encrypt(data, reqByteCount);
  char *frame = f->getFrameData();
  int bytesEmbedded = 0;
  while (bytesEmbedded<reqByteCount && offset<this->dec->getFrameSize()) {
    for (int j = 7; j >= 0; j--) {
      if ((((1 << j) & data[bytesEmbedded]) >> j) == 1) {
        frame[offset++] |= 1;
      } else {
        frame[offset++] &= ~1;
      }
    }
    bytesEmbedded++;
  }
  this->crypt->decrypt(data, reqByteCount);
  if (offset == this->dec->getFrameSize()) {
    return make_pair(bytesEmbedded, 0);
  } else {
    return make_pair(bytesEmbedded, offset);
  }
};
```

Listing 3.5: LSB embedding implementation (`steg/lsb_algorithm.cc:8`)

`images/lsb_ill.png`

Figure 3.2: Illustration of the LSB Embedding algorithm, taken from `forensicmag.com`

Using the LSB embedding algorithm, 43.2 kB of data is is embedded into a single 1280×720 video frame (using a capacity setting of 100%). The resulting stego image is shown in Figure 3.3, with the left image being the cover object and the right image the stego object.

The visual impact on the video frame is very small and almost certainly not noticeable having been reproduced within this document at a smaller resolution. However, if we take images/lsb_emb_full.png

Figure 3.3: Effect of the LSB Embedding algorithm.

a closer look at a specific portion of the video frame, we can see some small discrepancies between the cover and stego objects - see Figure 3.4. I am unsure how well these images will be reproduced when printed, but the difference is definitely noticeable within the `PDF`. Without the original cover object for comparison, it would be very hard to identify these details visually and deduce the presence of embedded data. However, if the LSB plane of the frame is visually inspected, as in Figure 3.5, one can immediately detect the presence of the hidden data - the LSB plane of the stego object looks “too random”. This is an example of a *visual steganalysis attack*.

Formalising the looking “too random” idea leads to the *Chi-Squared attack* developed by Westfeld and Pfitzmann[36]. To implement the Chi-Squared attack, the concept of *Pair of Values* (PoVs) is introduced.

One of the effects of an embedding algorithm like sequential LSB embedding is the creation of POVs, pixel values that embed into one another. For example, a pixel value of 100 in the cover image will either stay 100 or change to 101. Similarly, a pixel value of 101 will either stay 101 or change to 100. Thus (100, 101) is a POV.

Definition 3.1. Pairs of Values

A POV $\boldsymbol{p}$ *is a member of the set* $\mathcal{P}ov$*, defined as,*

$$\mathcal{P}ov \triangleq \{\ (2k, 2k+1) \mid 0 \leq k \leq 127\ \}$$

Westfeld and Pfitzmann claim that the LSBs in images are not completely random, rather, the frequencies of each of the two pixel values in each POV tend to lie far from the mean of the POV. That is, it is unlikely for the frequency of pixel value $2k$ to be close to equal to the frequency of pixel value $2k+1$. Furthermore, as information is embedded into the cover object, the frequencies of $2k$ and $2k+1$ become (nearly) equal. The Chi-squared attack was designed to detect this and bases the probability of embedding on how close to equal POVs are in the image.

images/closeup_3.png

Figure 3.4: Close up effect of the LSB Embedding algorithm.

images/lsb_emb.png

Figure 3.5: LSB plane of the cover and stego object.

To implement the attack, the following steps are taken in reference to a given image. First, $x_k = \text{frequency}(2k)$ and $y_k = \text{frequency}(2k+1)$ are calculated, followed by the expected frequency $z_k = \dfrac{x_k + y_k}{2}\ \forall\ k$. n is defined to be the number of POVs, $|\mathcal{P}ov|$. For uncompressed `AVI` using 24 bits per pixel, $n = 128$. The *minimum frequency condition* is now applied. This sets $x_k = y_k = z_k = 0$ and decrements n by one, if the condition $x_k + y_k \leq 4$ holds. The Chi-Squared statistic, with $n - 1$ degrees of freedom is then calculated:

$$\chi^2_{n-1} = \sum_{k=0}^{127} \frac{(x_k - z_k)^2}{z_k}$$

The probability of embedding, p, is then calculated by evaluation of the following integral:

$$p = 1 - \frac{1}{2^{\frac{n-1}{2}}\Gamma(\frac{n-1}{2})} \int_0^{\chi^2_{n-1}} e^{-\frac{u}{2}} u^{\frac{n-1}{2}-1} du$$

See Appendix Section C.1 for an implementation of the Chi-squared attack in Python.

The Chi-squared attack produces very good results for the naïve sequential LSB embedding algorithm. Figure 3.6 show an example of it in use operating on a video frame with a capacity setting of 50%. It obvious that information has been embedded within the first half of the frame.

`images/chi_graph.png`

Figure 3.6: Results of the Chi-Squared attack.

The Chi-Squared attack motivates the development of a *permuted* LSB embedding algorithm. The attack works so well because regardless of the capacity setting, the data is sequentially embedded from the top of the video frame. It would be better to distribute the data uniformly throughout the entire frame. A permuted LSB embedding algorithm achieves this and resists the Chi-Squared attack when using a small capacity setting[17].

[17] A permuted embedding using a capacity of 100% is no different to sequential embedding!

3.3.2 Permuted LSB Embedding

One can produce deceptively simple pseudocode for the permuted LSB embedding algorithm as shown in Algorithm 3.5.

Algorithm 3.5 Permuted LSB embedding algorithm

```
1: path ← a pseudorandom permutation of the cover object
2: path.seekToOffset(offset-1)
3: for i ← 0 upto dataBytes - 1 do
4:     for j ← 7 downto 0 do
5:         if The jth significat bit of data[i] == 1 then
6:             Set LSB(frame[path.next()]) to 1
7:         else
8:             Set LSB(frame[path.next()]) to 0
```

Lines 1 and 2 hide a large amount of complexity involved in implementing the algorithm.

Assuming the cover object consists of n bytes, a pseudorandom permutation of the cover object can be thought of as a pseudorandom permutation of the numbers 0 - n. One way to produce a pseudorandom permutation of a list of numbers is to shuffle an array containing them. This approach has the drawback that you need to hold the numbers in memory (11 MB for a single 720p HD video frame). Instead, I took a different approach using a *Linear Congruential Generator*[37] (LCG).

Definition 3.2. Linear Congruential Generator

A Linear Congruential Generator is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m$$

where X is the sequence of pseudorandom values, and

$$\begin{gathered} m,\ 0 < m \text{ - the modulus} \\ a,\ 0 < a < m \text{ - the multiplier} \\ c,\ 0 \leq c < m \text{ - the increment} \\ X_0,\ 0 \leq X_0 < m \text{ - the seed} \end{gathered}$$

are integer constants that specify the generator.

The pseudorandom permutation is only needed to uniformly distribute embedding throughout the image and it is therefore not a problem that a LCG may not produce cryptographically secure random numbers. The Hull-Dobell Theorem[38] states that a LCG will have a full period if and only if the following 3 requirements are satisfied:

1. $gcd(c, m) = 1$ (c and m are relatively prime),
2. $a - 1$ is divisible by all prime factors of m,
3. $a - 1$ is a multiple of 4 if m is a multiple of 4.

Therefore, if the above requirements can be satisfied, a LCG can be used as a pseudorandom permutation for the cover object with m equal to n (the size of the cover object). Note that forcing m to be a power of 2, simplifies the above requirements to:

1. $gcd(c, m) = 1$,
2. $a - 1$ is odd,
3. $a - 1$ is a multiple of 4 if m is a multiple of 4.

As such, m is set equal to the next power of 2 larger or equal to n. Any produced values larger or equal to n are just discarded, therefore producing a full period of size n as required.

The pseudorandom permutation should be dependant on a user provided passphrase. Therefore, the values of a and c are determined using a key derived from the users passphrase. The popular key derivation function `PBKDF2`[39] is used along with the `Whirlpool`[40] hash function to generate a 128 byte key pool and the first 4 bytes are taken as an unsigned integer and used to derive c and a. Rather than implement these well known algorithms myself, I used a popular C++ cryptographic library, `Crypto++`.

The result of the permuted embedding algorithm is illustrated in Figure 3.7. The left image has data embedded using the sequential embedding algorithm whereas the right image has the same data embedded using the permuted embedding algorithm. Both images were using a capacity setting of 15%.

`images/lsbp_ill.png`

Figure 3.7: Illustration of the permuted LSB algorithm.

Figure 3.8 shows the result of applying the Chi-Squared attack to the video frames in Figure 3.7. The left and right graphs show the probability of embedding within the left and right frame respectively. It is clear that the implemented permuted LSB embedding algorithm resists the Chi-Squared attack when using small values for the capacity setting. The above example embedded data within an uncompressed `AVI` video file which was 10 seconds in duration, had a resolution of 1280×720 and was 741 MB in size. Using the permuted LSB embedding algorithm with a capacity setting of 15% (which resisted the Chi-Squared attack), `Stegasis` informed me that the formatted volume had a total capacity of 14.49 MB, roughly 2% of the file size. This embedding capacity is already a images/lsbp_chi.png

Figure 3.8: Chi-Squared attack on the permuted LSB algorithm.

lot better than some of the those provided by programs investigated in Section 2.2, and will be further improved upon during the extension task within Section 3.6.1.

3.3.3 Encrypting the embedded data

To further strengthen the security of `Stegasis`, I decided to encrypt the data before embedding it within the video. This way, even if the stegosystem is broken, the information itself will not be compromised. The available cryptographic algorithms are decoupled from the steganogrphic systems allowing the user to mix and match between them. In addition, the same user provided passphrase used to permute the data throughout the video frame is used within the encryption process[18], addressing another concern of some of the investigated programs in Section 2.2.

The initial approach to encrypt the embedded data was to simply `XOR` it with a pseudorandom number stream. The `Crypto++` library provides a variety of suitable pseudorandom number generators which can be seeded. However, questions about the cryptographic strength of these number generators lead to the implementation of more standard encryption algorithms and the eventual removal of the pseudorandom number stream option. `AES` (256 bit), `TwoFish` and `Serpent` are all available within the `Crypto++` library which made incorporating them in `Stegasis` easy. Counter mode (`CTR`) is used as the block cipher mode so that data is not required to be padded to the block size. A hybrid chained encryption scheme `AES` → `TwoFish` → `Serpent` is also supported. This method has the advantage that if even 2 of the above algorithms are cryptographically broken, the data will still be secure[19]. You do however incur a performance penalty due to the encryption and decryption processes.

[18] A different section of the 128 bytes key pool is used.
[19] This encryption option is also offered by `TrueCrypt`.

This concludes the steganogrphic embedding algorithms developed to satisfy the core requirements operating on uncompressed `AVI` video. Table 3.1 shows a complete list of the implemented steganographic and cryptographic algorithms, including those implemented within the extension tasks (in italics). Although I have shown resistance to some steganalysis techniques, I have no doubt that there exist attacks that would break these implemented stego systems. However, more secure steganogrphic systems are investigated within the extension tasks and the framework `Stegasis` provides allows easy incorporation of new, more secure embedding algorithms.

Steganographic (Algorithm code)	**Cryptographic**
LSB Embedding (`LSB`)	AES
Permuted LSB embedding (`LSBP`)	Serpent
DCT LSB embedding (`DCT`)	TwoFish
Permuted DCT LSB embedding (OutGuess 0.1) (`DCTP`)	AES(Serpent(TwoFish))
F4 (`F4`)	
F5 (`F5`)	

Table 3.1: Implemented steganographic and cryptographic algorithms

3.4 The File system

I decided early on that the file system would only support a subset of the features offered by a fully-fledged standard file system. The following features were considered essential for a bare-bones practical file system: Creating and deleting files, reading and writing to files, listing the files in the file system and renaming (moving) files. Support for directories was also considered important and was therefore implemented as an extension task.

File permissions and access / modification times were considered non-essential and therefore not implemented. This means all files will be given the same permissions of `755` (`RWXRW-RW-`), and all access / modification times default to 0 *Unix Time* (1 January 1970).

Since the file system functionality was being implemented from scratch, the code written turned out complex and intricate due to the many corner cases encountered during integration testing. A large amount of time was dedicated to testing the implemented file system and tracking down particularly nasty concurrency bugs arising in specific circumstances.

The basic idea behind storing a file system within a video is to develop some kind of header which contains the locations of all of the files stored across the video frames. The files themselves will be broken into arbitrary sized *chunks* and stored within a particular frame at a particular offset. Consider the following example use case which motivates the solution developed in this section. A 10 byte text file is first written to the volume and embedded within frame 1 of the video at offset 0. A second file is now written to the volume and is such embedded within frame 1 at offset 80.[20] Now, additional text is appended to the first file, where should this be embedded? The obvious answer is after the second file, thus requiring some sort of header to keep track of the chunks so that the files can be read back correctly.

3.4.1 Developing the header

The header serves a similar purpose to the *File Allocation Table* used with the `FAT` file systems. It will need to be stored in a known location so that it can be extracted when the video file is mounted. I decided to use the first frame of the video to store the header; referred to as the *File Allocation Frame* (`FAF`). The design of the header went through a number of iterations before arriving at the final version presented here, mainly due to underestimating the number of bits needed to store the file frame numbers and offsets. Figure 3.9 shows the overall structure of the `FAF`.

images/header.png

Figure 3.9: File allocation frame structure.

The header section (coloured blue) contains metadata about the embedded file system. "STEG" is the literal 4 character `ASCII` string and is used to check the header has been correctly extracted. If STEG is not found as the first 4 bytes of the header, the extraction process is aborted and an error message displayed to the user. The "Header Bytes" field contains the number of remaining bytes to be extracted from the `FAF`. This data contains information for each of the files within the file system, including the file name as a null terminated string, the number of chunks the file is split into and the location and size

[20]Since it takes 8 bytes of frame to embed 1 byte of data.

of each of these chunks. A file chunk is represented as a `FileChunk` struct defined in Listing 3.6.

```
struct FileChunk {
  uint32_t frame;
  uint32_t offset;
  uint32_t bytes;
};
```

Listing 3.6: FileChunk definition (`fs/stegfs.h:19`)

Within Figure 3.9, a single file chunk is indicated with square brackets, and is coloured green. A file can contain an arbitrary number of chunks each of which can span multiple frames. Note that using an unsigned integer for the struct fields limits individual chunk sizes to roughly 4 GB. However, this shouldn't be a problem since writes to the file system occur in 65 kB chunks.

3.4.2 Writing to the file system

A write to the file system occurs in three main stages. First, `create` will be called requesting that a new file be created with a specified name. Next, the `write` function will be called a number of times requesting that data be written to the file. Finally `flush` will be called for the file requesting that any data held in memory be flushed to disk.

The first stage is easy to implement and is listed in Listing 3.7.

```
int SteganographicFileSystem::create(const char *path, mode_t mode, struct
    fuse_file_info *fi) {
  this->fileSizes[path] = 0;
  this->fileIndex[path] = std::vector<struct FileChunk>();
  return 0;
};
```

Listing 3.7: The `create` function call (`fs/stegfs.cc:202`).

The write calls are a bit trickier, recall the `FUSE` write call which requests that `size` bytes from the buffer `buf` should be written to the file `path` starting at offset `offset`. Algorithm 3.6 shows pseudocode for the write function. The full implementation is listed in the Appendix Section C.2.

Finally, `flush` is called which will write the header and ask the video decoder to write back to disk. This can either occur synchronously or asynchronously depending on a user specified flag.

3.4.3 Reading from the file system

The read function call is similar, recall the declaration which requests that `size` bytes of the file `path` starting at offset `offset` should be written to the buffer `buf`. The read function must identify the chunk `offset` points to and then return the correct amount of data possibly spread across multiple subsequent chunks. Algorithm 3.7 describes the function operation. See the Appendix Section C.2 for the full implementation.

Algorithm 3.6 Writing to the file system.

```
bytes_written ← 0
(next_frame, next_offset) ← decoder.getNextFrameOffset()
chunk ← FileChunk()
while bytes_written < size do
    (bytes_embedded, new_offset) = embed size-bytes_written from buf+bytes_written
into next_frame at next_offset
    chunk.bytes += bytes_embedded
    next_offset = new_offset
    if new_offset == 0 then
        next_frame ++
    bytes_written += bytes_embedded
fileIndex[path].append(chunk)
return size
```

Algorithm 3.7 Reading from the file system.

```
bytesWritten ← 0, chunkNum ← 0, bytesWritten ← 0
for chunk in fileChunks do
    if bytesRead + chunk.bytes > offset then
        break
    else
        bytesRead += chunk.bytes
        chunkNum ++
while bytesWritten < size do
    chunk ← fileChunks[chunkNum]
    bytesLeftInChunk ← chunk.bytes - chunkOffset
    bytesLeftInChunk ← min(bytesLeftInChunk, size-bytesWritten)
    extract(chunk.frame, chunk.offset + chunkOffset, bytesLeftInChunk,
buf+bytesWritten)
    bytesWritten += bytesLeftInChunk
    chunkOffset ← 0
    chunkNum ++
```

3.4.4 Listing files in the file system

The function `readdir` is called when the contents of a directory are requested to be listed, the implementation is straightforward and just iterates of the fileSizes map as shown in Listing 3.8.

```
int SteganographicFileSystem::readdir(const char *path, void *buf,
    fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi) {
  filler(buf, '.', NULL, 0);
  filler(buf, '..', NULL, 0);
```

```
  for (auto kv : this->fileSizes) {
    filler(buf, kv.first.c_str() + 1, NULL, 0);
  }
  return 0;
};
```

Listing 3.8: FUSE readdir implementation (fs/stegfs.cc:264).

The above FUSE operations cover the majority of the core file system implementation, successfully implementing all of the decided essential features listed at the start of this subsection.

3.5 Command line application

To complete Stegasis, the implemented components must be assembled together and a user interface produced. I decided on a command line interface allowing the user to utilise the functionality developed within this project. Listing 3.9 shows the process of formatting and mounting a video using Stegasis.

```
$ stegasis format --alg=lsbp --crypt=aes --pass=123 --cap=25 ~/my_video.avi
Formatting video...
Volume size: 30 MB
Format Successful
$ stegasis mount --alg=lsbp --crypt=aes --pass=123 ~/my_video.avi /mnt/video
Video mounted at /mnt/video
```

Listing 3.9: Using Stagasis to format and mount a video.

At this point, the user can copy files into /mnt/video and they will be automatically embedded within the video. The user can unmount the video by closing the program (for example by pressing Control-C), Stegasis will gracefully exit, writing back any unflushed changes to disk. See the Appendix Section B for more detailed usage information.

Stegasis works exactly how I envisioned it during the project inception. The example usage I gave within the project proposal is exactly reflected within the finished product[21].

3.6 Extension Tasks

All of the extension tasks listed within the project proposal were addressed and with the exception of "Hiding data within audio streams", all of them were successfully implemented. The further extension task "Plausible deniability" was added during the project and also successfully implemented.

3.6.1 Supporting multiple video formats

As discussed, there are several reasons why it would be beneficial for Stegasis to operate on video formats other than uncompressed AVI. To accomplish this, rather than develop

[21] With the exception of pressing Control-C to exit rather than typing stegasis unmount.

many video decoders and steganographic systems for multiple video formats, a generic solution was designed and implemented utilising `FFmpeg`.

Regardless of the video format, a video can be thought of as a sequence of video frames played back at a specific frame rate together with an optional audio track. Using `FFmpeg` to extract the frames of a video and implementing steganographic systems operating on sequences of images, `Stegasis` can function seamlessly across multiple video formats. For example, if a user provides a compressed `h.264` video, `FFmpeg` will extract each frame as a `JPEG` along with the audio track as an `mp3`. `Stegasis` will then embed and extract data into and out of the extracted `JPEG` images. When the video is unmounted, the modified images will be re-muxed as a *Motion JPEG* stream together with the audio to produce an `MKV` file - the stego object. The `JPEG` file format was chosen for the intermediate frames due to the large number of steganographic algorithms operating on them. Figure 3.10 shows an overview of the approach.

images/multi.png

Figure 3.10: Operation on multiple video formats.

This method will work with virtually any video format presented to `Stegasis` due to the extensive codec library of `FFmpeg`.

Although the initial conversion of the provided video to `JPEG` images can be lossy, the reassembly of video frames and extraction thereafter cannot - this would damage the embedded file system. This constraint leads to the following four `FFmpeg` commands shown in Listing 3.10.

```
// Convert an arbitrary video to JPEG images (Lossy)
ffmpeg -r <fps> -i <path> -qscale:v 1 -f image2 /tmp/stegasis/image-%d.jpg
// Extract video audio
ffmpeg -i <path> /tmp/stegasis/audio.mp3
```

```
// Losslessly convert modified JPEG images to an MJPEG stream
ffmpeg -framerate <fps> -i /tmp/stegasis/image-%d.jpg -i /tmp/stegasis/
    audio.mp3 -codec copy -shortest output.mkv

// Losslessly extract JPEGs from a video already formatted by Stegasis
ffmpeg -r <fps> -i <path> -vcodec copy /tmp/stegasis/image-%d.jpg
```

Listing 3.10: `FFmpeg` frame extraction and reassembly command.

The first frame extraction command converts each frame of the video file to a `JPEG` image using the highest quality setting and writes them to `/tmp/stegasis/`. The reassembly command muxes the modified video frames and audio together into a single `MKV` file. The use of `-codec copy` tells `FFmpeg` to literally copy the `JPEGs` into a Motion JPEG (`MJPEG`) stream. This ensures the frames are not recompressed - preserving the embedded data. This process was verified to be correct by noting the `MD5` hash of the `JPEG` frames prior to being muxed. These frames were then extracted from the resulting `MKV` file, their hashes computed and compared. The second frame extraction command uses `-vcodec copy` which tells `FFmpeg` to losslessly extract each `JPEG` image from the `MJPEG` stream - preserving the embedded file system.

Using the above commands, I implemented a video decoder performing the described method. When `Stegasis` is run, a check occurs to see whether the provided file is an uncompressed `AVI`. If it is, the native `AVI` decoder developed is used. If it is not, this video decoder is used, allowing `Stegasis` to operate seamlessly across all video types without the user needing to manually specify which video decoder to use.

The final step of this extension task is to implement embedding algorithms operating on the extracted `JPEG` images. Due to the design of `Stegasis`, by implementing the new algorithms satisfying the `Steganographic Algorithm` interface, all of the file system logic will continue to work as expected.

The implemented algorithms all embed data within the least significant bits of the `JPEGs` discrete cosine transform coefficients. As with the core project, several versions of the basic algorithm were implemented along with more advanced algorithms such as `F4` and `F5`. Algorithm 3.8 shows pseudocode for the basic embedding algorithm.

Algorithm 3.8 Basic `JPEG` embedding algorithm.

```
for i ← 0 upto data_bytes - 1 do
    for j ← 7 down to 0 do
        (row, block, coefficient) ← getCoefficientForOffset(offset++)
        component = 2             ▷ Components 2 and 3 are the chroma components.
        row ← frame.getRow(row, component)
        if The jth significant bit of data[i] == 1 then
            Set LSB(row[block][coefficient]) to 1
        else
            Set LSB(row[block][coefficient]) to 0
```

Within the actual implementation, the component to embed within is chosen using the calculation `(co % 2) + 1`. This uniformly distributes the embedded data bits between the two chroma components, deliberately not touching the luminance component.

At higher capacity settings, this embedding algorithm does begin to produce visual artifacts within the video frames. Figure 3.11 illustrates this when a capacity setting of 100% is used.

images/bb_100cap.png

Figure 3.11: Embedding artifacts at 100% capacity.

However, at lower capacity settings it is not possible to visually differentiate between the cover and stego objects[22]. Roughly, any capacity setting above 50% begins to introduce visually detectable artifacts. Figure 3.12 shows the same frame as above, but using a capacity setting of 20%. (Left image is the original frame.)

images/bb_20cap.png

Figure 3.12: Frame comparison at 20% capacity.

The basic `JPEG` embedding algorithm and the permuted variant (`OutGuess 0.1`) are both vulnerable to the *histogram attack* discussed in Section 4.2. This motivates the implementation of more advanced algorithms such as `F4` and `F5` which resist this type of steganalysis attack. Instead of replacing the LSB of DCT coefficients, `F5` decrements the coefficients absolute value via *matrix encoding*. This results in no pairs of values being produced and thus `F5` cannot be detected by the Chi-Squared attack. Matrix encoding within `F5` uses *Hamming codes* to embed a k-bit message into a $2^k - 1$ bit code word,

[22]This claim is verified within the evaluation Section 4.2.

changing it at most by one bit. This allows F5 to achieve high embedding efficiencies. See the Appendix Section C.3 for more details and an implementation of F5.

Due to the final lossless stage of JPEG compression, it can in some cases[23] be possible to embedded files larger than the cover object - achieving embedding capacities in excess of 100%. This is because the encoding effectively compresses the embedded data before it is written back to disk. To show this in action, Listing 3.11 shows a 306 MB file embedded within a 189 MB video.

```
$ ls -lah
-rw-rw-r-- 1 scott scott 189M Jan 15 22:30 video.mkv
$ stegasis mount -alg=dctl video.mkv /mnt/video
[...]
Mounting...
$ ls -lah /mnt/video
total 4.0K
drwxr-xr-x  2 root root    0 Jan  1  1970 .
drwxrwxrwt 12 root root 4.0K Jan 15 22:26 ..
-rwxr-xr-x  1 root root 306M Jan  1  1970 file1
```

Listing 3.11: Demonstration of 162% embedding capacity.

The performance of Stegasis operating on JPEG images is noticeably slower compared to uncompressed AVI files. This is due to the necessary decompressing and recompressing of the JPEG frames. As mentioned in the preparation section, it is not possible to hold all of the decompressed JPEG frames in memory, meaning they will need to remain compressed, only being decompressed when requested. Initially, the JPEG files were left on disk and read into memory and decompressed when requested before being recompressed and written back. This understandably gave terrible performance due to the large amount of disk IO. This was rectified by reading all of the compressed JPEG images into memory upon video mount. The decompression and recompression operates could then operate on this memory - no longer involving any disk IO.

The above described additions to Stegasis allow it to seamlessly operate across a large range of video formats, greatly increasing practicality and successfully completing the extension task.

3.6.2 File system directory structures

The core implementation of the file system does not allow the creation and manipulation of directories, forcing all files written to the volume to reside in the root. Although Stegasis is still usable with this limitation, it would be nice to allow users to organise their embedded files using directories as you would expect from a normal file system. To achieve this, the mkdir FUSE operation will need to be implemented along with a few changes made to the current file system implementation.

[23]This is highly dependant on the data being embedded. The best possible scenario is embedding all zeros.

A third data structure, `dirs` is first added along side `fileSizes` and `fileIndex` containing each of the directories within the file system. The `mkdir` operation is then trivial to implement as shown in Listing 3.12.

```
int SteganographicFileSystem::mkdir(const char *path, mode_t mode) {
  this->dirs.insert(path);
  return 0;
}
```

Listing 3.12: `FUSE` mkdir implementation (`fs/stegfs.cc:158`).

The `readdir` implementation will need to be modified to correctly list the files with respect to the current working directory. It is no longer correct to just iterate over all files in the file system and return their names since you only want to return a file if the function call is requesting the folder that file directly resides within. Consider the following example wherein the file system contains one sub-directory and two files; `/test.txt`, `/folder/other.txt`. If `readdir` requests a path of `/folder/`, only `/folder/other.txt` should be returned. This can be accomplished by testing if the requested path is a prefix of the file name. However, this method fails when directories are created within directories - files within sub-directories of the path should not be returned. This is fixed by checking the number of slashes in the file name and comparing this number of slashes in the path. Algorithm 3.9 describes the `readdir` implementation.

Algorithm 3.9 Algorithm for the `readdir` implementation.

```
add('..')
add('.')
pathSlashes ← number of slashes in path
for for file in (fileSizes and dirs) do
    fileSlashes ← number of slashes in file
    if path == / then
        if file contains one slash then
            add(file)
    else if path is a prefix of file and pathSlashes == fileSlashes - 1 then
        add(file)
```

Changes will now need to be made to write the directories to the video file and to read them back, preserving the directory structure between unmounts and remounts. I chose to represent a directory as a file in the header of the video which has a value of -1 in the `number of triples` field. This meant the structure of the `FAF` did not need to be modified. The `readHeader` and `writeHeader` functions were modified appropriately.

These changes successfully implement the extension task and greatly improve the usability of the file system. With the exception of permissions and access and modification times, `Stegasis` now provides all the functionality one would expect from a typical file system.

3.6.3 Plausible deniability

Similar to `TrueCrypt`'s hidden volume feature, I planned to implement plausible deniability by embedding two separate file systems within one video, each using a different user provided passphrase. The *outer* volume will reside at the beginning of the video (where it usually would) and the user should populate this volume with files they are willing to reveal if necessary. The *inner* volume will reside half way through the video file and the user should place sensitive files here. When forced to give up the encryption keys, the user can reveal the passphrase for the outer volume in confidence the inner volume will not be compromised.

This method is vulnerable to steganalysis attacks that would be able to detect the presence of the hidden volume half way through the video. To combat this, random data is embedded throughout the entire video during the format process using both passphrases. Since the volumes are encrypted, it will not be possible to tell if the identified embedded data is a hidden volume or just the random data written during the formatting process[41].

The modifications required for `Stegasis` to implement this feature were surprisingly simple. A second passphrase command line flag `pass2` was added, if this is specified during the format process two headers are written at the start and in the middle of the video. The mount process now attempts to extract and decrypt the first header using the provided passphrase. If this fails, it then attempts to decode the second header (embedded within the middle frame). Depending on which header extracts successfully, the outer or hidden file system is presented to the user. Listing 3.13 shows an example use case of the plausible deniability functionality.

```
$ stegasis format --alg=lsb --crypt=serpent --pass=outer --pass2=hidden --cap=20
    video.avi
[...]
Format successful!
$ stegasis mount --alg=lsbp --crypt=serpent --pass=outer video.avi /tmp/steg
[...]
Mounting...

[Second Terminal]
$ echo 'outer' > /tmp/steg/outer.txt

[First Terminal]
<Control-C>
[...]
$ stegasis mount --alg=lsbp --crypt=serpent --pass=hidden video.avi /tmp/steg
[...]
Mounting...

[Second Terminal]
$ echo 'inner' > /tmp/steg/sensitivefile.txt
```

Listing 3.13: `Stegasis` plausible deniability functionality.

Since the header and data for the hidden file system is stored within the middle video frame onwards, it is possible to damage it by writing too much data to the outer volume. The user has effectively sacrificed half of the total embedding capacity in return for plausible deniability.

3.6.4 Hiding data within audio

It is stated within the project proposal that "a substantial part of an `AVI` file may be the audio data". I was correct to use the word "may" within this statement since it turns out that a very small percentage of an uncompressed `AVI` file is the audio data. Similarly low percentages apply to other video formats due to modern audio compression algorithms such as `MP3`. The question of how this extra embedding capacity could actually be utilised is also quite prevalent since `Stegasis` and the file system logic all rely on the idea of the video being broken up into frames. This concept does not translate easily over to audio. I therefore decided not to pursue this extension task.

3.6.5 Evaluation of embedding impact on video quality

To evaluate the requirement "Embedding should occur with no perceivable impact on video quality.", a web application was required to be developed and hosted. Figure 3.13 shows a screenshot of the finished website hosted at `www.stegasis.co.uk` for the majority of the duration of the project.

images/site.png

Figure 3.13: Screen shot of the evaluation study site.

The user is presented with two images and asked the question "Which image do you think contains hidden data?". The user may inspect the image for as long as they wish, and by clicking on an image, they then select their choice.

In total there are 14,912 pairs of images, one of which is randomly selected for each page load. One of the images is the original video frame while the other has 2.5 kB of data embedded within it using the permuted JPEG embedding algorithm and a capacity setting of 20% (the default value). The position of the "correct" image is randomly chosen and the file names of the images are also random strings. This tries to thwart any attempts at foul play, possibly damaging the collected results. When a user first visits the site, a cookie containing a unique ID is issued allowing data points from the same user to be aggregated together.

The results of this evaluation study obtained by the implemented web application are discussed in Section 4.2 and conclude that it is not possible to visually differentiate between the cover and stego objects at the default capacity setting of 20%.

3.7 Testing

Stegasis was tested using a combination of unit, integration and visual testing. Due to the decoupled nature of the steganogrphic embedding algorithms, it was easy to produce unit tests for the embedding and extraction functionality. A total of *largenumber* test cases were written testing all stenographic algorithms and associated code. Listing 3.14 gives an example unit test.

```
bool SteganographicUnitTests::completeness(SteganographicAlgorithm alg) {
    char *in, *out; this->initRandomData(in, out);
    alg->embed(this->dummyChunk, in, this->dataSize, 0);
    alg->extract(this->dummyChunk, out, this->dataSize, 0);
    return compareData(in, out);
};
```

Listing 3.14: Embedding algorithm unit test (`steg/steg_algorithm_test.cc:100`).

The above unit test verifies the steganogrphic correctness of the stego systems, checking that data embedded using a key **k** is correctly extracted when using the same key.

I deemed it more appropriate to test the file system functionality via an integration test suite due to its highly coupled nature. The test suite revolved around a number of compressed archives containing test files and directory structures. Stegasis is first instructed to mount a test video into /tmp/test. These archives are then copied into the file system and uncompressed. Once the extraction process has completed, the resulting file system is traversed to check the contents is as expected.

Visual testing was also employed. For example, large media content such as high definition video was copied into the mounted volume and checked to see if it played back correctly. This is a good test case since the file (due to its size) will be spread across a large proportion of the video. Indeed, several bugs were identified using this testing approach.

The AVI parser was tested using a black box testing approach on a number of different uncompressed AVI videos. The parser prints out debug information about the video file which can be visually inspected. For example, the resolution of the video contained within the `BITMAPINFOHEADER` can be compared against the known value. Checking that the `AVI` still plays back after steganographic modification is also a good indication that the parser is operating successfully.

See the Appendix Section D for some more testing code samples.

4 || Evaluation

4.1 Satisfaction of Requirements

`Stegasis` satisfies all of the core requirements and addresses all of the proposed extension tasks. I therefore consider the project a success. Each requirement will now be addressed in turn.

4.1.1 Embedding data within video files

`Stegasis` does indeed allow data to be embedded within video files. Furthermore, it provides a total of 6 different steganogrphic embedding algorithms all of which satisfy correctness[24] and, an evaluation study discussed in Section 4.2 below concludes that `Stegasis` can operate with no perceivable impact on video quality. A number of steganalysis tools were developed to test the security of the proposed steganographic systems, prompting the design and implementation of more secure embedding algorithms. Encryption functionality is also provided making use of the user provided passphrase. Finally, a capacity setting was also implemented allowing users to choose a trade off between steganographic security and embedding capacity.

4.1.2 Providing a file system interface

`Stegasis` allows users to specify a mount point at which the volume will be mounted and data written to and read from this volume will embedded and extracted from the video on the fly. Standard Unix tools including `cp`, `mv` and `rm` work as expected and the file system is correctly persevered between unmounts and remounts of the same video.

4.1.3 Supporting uncompressed AVI

Raw uncompressed `AVI` files are supported via a developed native parser which allows access to and modification of individual pixel data along with extraction of meta-data about the video.

4.1.4 File system performance

The performance of the file system allows full HD video content to be played back from directly within the volume (12 Mb/s). However, achieving read and write speeds in excess of 20 MB/s was not a trivial task. The file system can provide read and write speeds in excess of 30 MB/s (and under ideal conditions[25], 80MB/s.), but only for the `AVI` algorithms which do not encrypt the embedded data and only if the performance flag has been specified (to stop synchronous flushes to disk). Performance is discussed in more detail in Section 4.3.

[24] Verified by unit test cases as explained in Section 3.7.
[25] These conditions are discussed in Section 4.3.

The extension task requirements are now discussed individually.

4.1.5 Supporting multiple video formats

The novel method described in Section 3.6.1 using `FFmpeg` allows `Stegasis` to seamlessly operate across virtually every video format available today. The basic implemented embedding algorithms operating on `JPEG` images (`DCT`, `DCTP`) offer large embedding capacities allowing multiple large files to hidden inside of a single video, while the more advanced implemented algorithms (`F4`, `F5`) offer high steganographic security.

4.1.6 File system directory operations

Directory functionality was successfully implemented allowing the creation and manipulation of directory structures, enabling users to organise their embedded files within folders. The `mv` and `rm` commands work as expected when operating on directories as well as normal files.

4.1.7 Embedding within audio

This extension task was investigated but I decided not to pursue it due to the relativity little embedding capacity gain compared to the work required to implement it.

4.1.8 Plausible deniability

Plausible deniability functionality was implemented allowing a second hidden file system to be optionally embedded within a video. Depending on the passphrase provided by the user during mounting, either the outer or inner volume is presented. This satisfies the first requirement point. The second point, that the hidden volume should not be detectable, has also been satisfied since it is not possible to differentiate between the random data written during the format process and the second encrypted hidden volume[41].

4.1.9 Evaluation of the visual impact of embedding

A web application was implemented and hosted for the majority of the duration of the project. A total of 2040 data points were collected from 21 unique users. The results are discussed below in Section 4.2, and the outcome of the study confirmed that it was not possible for the users to visually perceive embedding artifacts produced by `Stegasis` when using the default capacity setting (20%). This satisfies the final extension task.

`Stegasis` therefore has addressed all of the proposed extension tasks, further cementing the success of the project.

4.2 Security

Although security of the steganographic algorithms was not a major focus of this project, steganalysis techniques were implemented and these led to more secure algorithms being developed. The evaluation user study was also focused on security - attempting to decide if it is possible to visually differentiate between cover and stego objects produced by `Stegasis` when using its default capacity setting. The results of this study are now statistically analysed.

Due to the layout of the website, if a user cannot tell the difference between the two presented images, they will select one arbitrarily. From the phrasing of the question posed to the user, a response is considered correct if the image selected did contain hidden data and incorrect if it did not. If a user were to randomly guess each time, the resulting data stream would be a random stream of corrects and incorrects. Letting correct be represented by 1 and incorrect by 0, this stream of random corrects and incorrects becomes a random bit string. Therefore, if it can be shown that the collected data is a random bit string, it can then be concluded that the users must have been randomly guessing and therefore could not differentiate between the two presented images.

The raw bit string obtained from the user study is included in the Appendix Section E. To determine if the obtained bit string is in fact random, firstly a hypothesis test is constructed to test if the bits are i.i.d. Bernoulli(1/2) which would be the case for a truly random bit string.

Let the bit string $\mathbf{X} = X_1 \dots X_n$, with each $X_i \overset{iid}{\sim}$ Bernoulli(p) define $T = \sum_{i=1}^{n} X_i$. T is a sufficient statistic for p. Clearly $T \sim$ Binomial(n, p).

Test $H_0 : p = \frac{1}{2}$ against $H_1 : p \neq \frac{1}{2}$ at significance level α. Under H_0, $T \sim$ Binomial$(n, \frac{1}{2})$, by the de Moivre-Laplace theorem $T \sim N(\frac{n}{2}, \frac{n}{4})$ for large n. So $(1-\alpha)$ confidence interval for T is

$$\left[\frac{n}{2} - z_{\alpha/2}\sqrt{\frac{n}{4}},\ \frac{n}{2} + z_{\alpha/2}\sqrt{\frac{n}{4}} \right].$$

For the bit string observed from the user study, $n = 2040$ and $T = 1042$. At $\alpha = 5\%$, $z_{\alpha/2} = 1.9600$, the confidence interval is $[975, 1065]$ which contains T. Therefore there is insufficient evidence to reject the null hypothesis and therefore insufficient evidence to conclude the bit string is not random.

Secondly, a number of statistical randomness tests are used. Both `ent`[42], a pseudorandom number sequence test program is used. Listing 4.1 shows the output of `ent` when applied to the bit string.

```
Entropy = 0.999664 bits per bit.

Optimum compression would reduce the size
of this 2040 bit file by 0 percent.

Chi square distribution for 2040 samples is 0.95, and randomly
would exceed this value 33.00 percent of the times.

Arithmetic mean value of data bits is 0.5108 (0.5 = random).
Monte Carlo value for Pi is 3.523809524 (error 12.17 percent).
Serial correlation coefficient is −0.035776 (totally uncorrelated = 0.0).
```

Listing 4.1: Output of `ent`.

These results strongly indicate that the bit string is random.

These analyses show that the bit string with high probability is random and therefore that the users could not differentiate between the cover and stego objects produced by `Stegasis`.

Although we have concluded that `Stegasis` can operate with no perceivable visual impact on video quality, this does not imply that it will resist further steganalysis techniques. For example, the *Histogram attack* which operates on `JPEG` images breaks the basic embedding algorithms (`DCT` and `DCTP`) implemented in the multiple video formats extension task (even using low capacity settings). The histogram attack enumerates the frequencies of the DCT coefficients of a given image. A characteristic of unmodified `JPEG` images is that its histogram is symmetrical about 0, that is, coefficients n and $-n$ have roughly equal frequencies. This characteristic is lost when embedding occurs. Figure 4.1 shows the effectiveness of the attack, comparing the histogram of an original video frame to one with embedded data. It clearly manages to differentiate between the cover and stego objects.

images/hist.png

Figure 4.1: The histogram attack on the `DCTP` algorithm.

`F4` and `F5` however resist this attack and provide a much higher level of steganographic security. Jessica Fridrich et al. showed they were able to break the `F5` algorithm by estimating the cover image histogram by slightly cropping the stego image. However, `Stegasis` is able to spread the embedded data across thousands of `JPEG` images using a small capacity setting, so I am unsure how effective this attack would be.

The framework `Stegasis` provides makes is easy to implement more secure embedding algorithms which would be a possible future project direction.

4.3 Performance

The performance of `Stegasis`, and in particular the file system, is very important to consider. As with `TrueCrypt`, using `Stegasis` should constitute a similar experience to copying files onto removable storage.

4.3.1 Performance of the file system

Due to the large embedding capacities offered, large files are expected to be copied into the file system and therefore embedded into the video. The write performance of the file system was therefore given a lot of attention during the implementation. The biggest improvement in file system performance was obtained by enabling the `FUSE` `big_writes` mount option. By default, `FUSE` limits write calls to blocks of 4096 bytes, `big_writes` allows this limit to be increased to 65 kB. Figure 4.2 shows the effect of varying the block size on write performance for the LSB embedding algorithm.

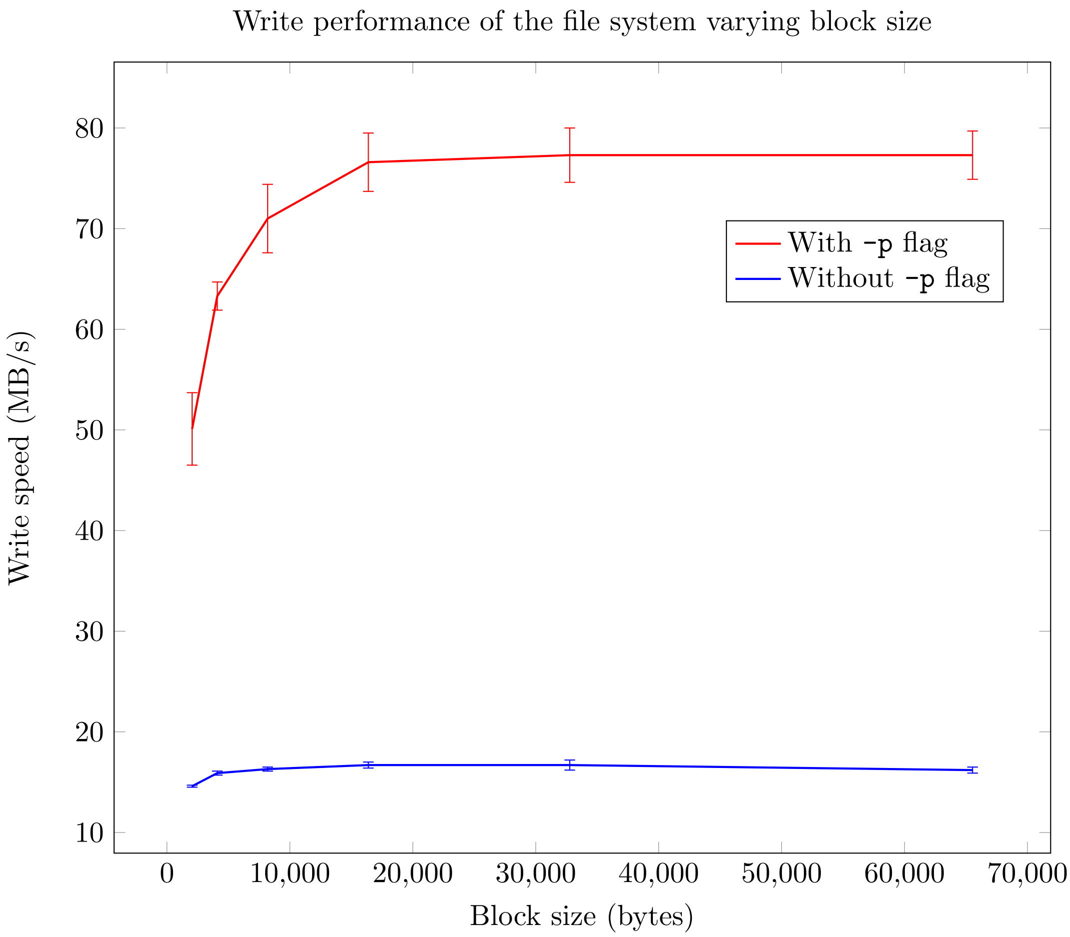

Figure 4.2: Write performance of the file system.

The data points for the graph were obtained using the command line tool `dd` as shown in Listing 4.2. Note that this test provides ideal conditions, having a large block size and writing all zeros - most use cases will not be like this.

To evaluate the file system performance under more standard conditions, I timed the copying of an 80 MB video file into and out of the file system for the LSB algorithm

```
$ dd if=/dev/zero of=test bs=65k count=1221
610+0 records in
610+0 records out
79953920 bytes (80 MB) copied, 0.456444 s, 77.1 MB/s
```

Listing 4.2: Testing the file system performance using `dd`.

using the `time` tool. This process was repeated multiple time and an average taken giving an average write speed of 33.6 MB/s and read speed of 46.3 MB/s. This is slower than above, but is still on par with USB 3.0 device speeds[43], surpassing the performance requirement. As expected, the embedding algorithms which permute and encrypt data give worse file system performance as shown in Table 4.1, but I think this is a fair trade off.

Stego System	**Encryption Algorithm**	**File Size (MB)**	**Copy time (s)**
LSB	AES	6.146	123
LSB	TwoFish	6.146	123
LSB	Serpent	6.146	123
LSB	AES(Serpent(TwoFish))	6.146	123
LSBP	AES	6.146	123
LSBP	TwoFish	6.146	123
LSBP	Serpent	6.146	123
LSBP	AES(Serpent(TwoFish))	6.146	123
DCT	AES	6.146	123
DCT	TwoFish	6.146	123
DCT	Serpent	6.146	123
DCT	AES(Serpent(TwoFish))	6.146	123
DCTP	AES	6.146	123
DCTP	TwoFish	6.146	123
DCTP	Serpent	6.146	123
DCTP	AES(Serpent(TwoFish))	6.146	123
F4	AES	6.146	123
F4	TwoFish	6.146	123
F4	Serpent	6.146	123
F4	AES(Serpent(TwoFish))	6.146	123
F5	AES	6.146	123
F5	TwoFish	6.146	123
F5	Serpent	6.146	123
F5	AES(Serpent(TwoFish))	6.146	123

Table 4.1: Steganographic and cryptographic algorithm combination performance.

4.3.2 Performance of video formatting

Another area worth considering is the time `Stegasis` takes to format video files. Although it is hard to quantify a "good" format time, I would expect a time of under 1 minute to be reasonable for an average 5 minute video. The format time of `AVI` files is more dependent on disk read performance, whereas for other video formats, processor performance will be more important (due to the video transcoding involved). I measured the time taken to format a variety of video files on my desktop computer. The results are summarised in table 4.2 below.

Video file	Volume capacity (MB)	Format Time (seconds)
700 MB 10 second `AVI`	96.60	6.146
14 MB 3 minute `MP4`	28.98	13.143
34 MB 4 minute `MP4`	159.07	29.270
80 MB 18 minute `FLV`	55.63	45.810

Table 4.2: Video format times

5 || Conclusions

Informally, I wanted this project to result in the "TrueCrypt of video steganography" focusing on a practical application allowing multiple files to be easily steganographically embedded within video files utilising a file system interface. This involved researching multiple technical fields (mainly Steganography, file systems and video formats) and developing an application to combine them all into a polished tool.

A number of steganographic embedding algorithms were researched and implemented borrowing ideas and algorithms from cryptography to further increase security. A file system in user space residing within a video file was designed and implemented offering the majority of the functionality one would expect from a standard file system. A native AVI parser was developed along with a novel method of supporting other video formats utilising FFmpeg. Finally a Linux command-line application, Stegasis, was produced combining the above into a single application.

Stegasis satisfies all of the core project requirements and implements all of the proposed extension tasks. In reference to the evaluation section of this document, I very much consider this project a success and hope to release Stegasis in the near future.

5.1 Lessons Learnt

The "Launch early, iterate often" approach taken to development did come with some drawbacks. Most notably, design decisions made early on during development were sometimes not given as much thought as they possibly should have. This resulted in some issues arising much later during development which required a lot of work to correct. Had due thought been given to these decision at the start of the development process, a large amount of time spent refactoring code could have been saved.

5.2 Future Project Directions

As discussed in the evaluation section, Stegasis currently lacks secure steganographic embedding algorithms. Further work would therefore likely involve implementing more secure embedding algorithms. Another avenue to explore would be implementing more native video decoders. For example, a native MP4 decoder could be produced which embedded data within motion vectors. This has an advantage over the FFmpeg method since a new, MKV video is not produced - the MP4 would be modified in place as occurs with AVI files. Finally, it would be great for Stegasis to be cross platform. Developing a Linux only application severely limits the target audience. However, since the file system was implemented using FUSE (which is not compatible with Windows) this may prove tricky.

References

[1] *Steganography in Digital Media*, Jessica Fridrich 2010

[2] *Regulation of Investigatory Powers Act 2000*
www.legislation.gov.uk/ukpga/2000/23/section/53

[3] *A Survey of Steganographic and Steganalytic Tools for the Digital Forensic Investigator*, Pedram Hayati et al.
www.pedramhayati.com/images/docs/survey_of_steganography_and_steganalytic_tools.pdf

[4] *TrueCrypt is a source-available freeware utility used for on-the-fly encryption.*
www.truecrypt.org

[5] *The NSA Thinks You Are an Extremist If You Care About Privacy*, Fahmida Rashid 2014, PCMag
securitywatch.pcmag.com/privacy/325273-the-nsa

[6] *The prisoners' problem and the subliminal channel*, Gustavus Simmons 1984, Advances in Cryptology, pp. 51 - 67

[7] *La Cryptographie Militaire*, Auguste Kerckhoffs 1883

[8] *Steganalysis: The Investigation of Hidden Information*, Neil Johnson and Sushil Jajodia 1998

[9] *Using Video Data*, ATLAS.ti 6 2011
atlasti.com/wp-content/uploads/2014/05/video_formats.pdf

[10] *AVI RIFF File Reference*, Microsoft
msdn.microsoft.com/en-us/library/windows/desktop/dd318189\%28v=vs.85\%29.aspx

[11] *Steganography and Steganalysis of JPEG Images*, Mahendra Kumar
www.cise.ufl.edu/~makumar/proposalppt.pdf

[12] *JSteg: Steganography and Steganalysis*, Murali P 2009
csis.bits-pilani.ac.in/faculty/murali/netsec-09/seminar/refs/anuroopsrep.pdf

[13] *Hide and Seek: An Introduction to Steganography*, Niels Provos and Peter Honeyman 2003, p. 36
niels.xtdnet.nl/papers/practical.pdf

[14] *F5 A Steganographic Algorithm*, Andreas Westfeld
www2.htw-dresden.de/~westfeld/publikationen/f5.pdf

[15] *Libjpeg, a widely used C library for reading and writing JPEG image files*
`libjpeg.sourceforge.net`

[16] *Human eye sensitivity and photometric quantities*
`www.ecse.rpi.edu/~schubert/Light-Emitting-Diodes-dot-org/Sample-Chapter.pdf`

[17] *FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video*
`www.ffmpeg.org`

[18] *FFmpeg Codecs Documentation*
`www.ffmpeg.org/ffmpeg-codecs.html`

[19] *FUSE: File system in userspace*
`fuse.sourceforge.net`

[20] *Linus vs FUSE: Kernel file system vs FUSE*, Sage 2011
`ceph.com/dev-notes/linus-vs-fuse`

[21] *Stegostick: A steganographic Tool that lets you hide any file into any file*, 2008
`sourceforge.net/projects/stegostick`

[22] *Stegomagic: An encrypting software application designed specifically for helping you hide files or messages in media items*, 2014
`www.softpedia.com/get/Security/Encrypting/StegoMagic.shtml`

[23] *Real steganography with TrueCrypt*, Martin Fiedler 2011
`keyj.emphy.de/real-steganography-with-truecrypt`

[24] *Stegovideo: A unique tool for hiding information in video*, Oleg Petrov 2011
`www.compression.ru/video/stego_video/index_en.html`

[25] *OpenPuff: A steganography tool supporting many carrier formats*, EmbeddedSW
`embeddedsw.net/OpenPuff_Steganography_Home.html`

[26] *Steganosaurus is a dissertation project exploring the application of video steganographic and video steganalysis techniques.*, James Ridgway 2014
`www.steganosaur.us`

[27] *YouTube Advanced encoding settings.*, Google 2014
`support.google.com/youtube/answer/1722171?hl=en-GB`

[28] *Example USB 2.0 flash drive*, Ebuyer 2015
`www.ebuyer.com/543103-toshiba-transmemory-8gb`

[29] *JavaFUSE provides Java bindings for FUSE*, Aditya Rajgarhia 2010
`code.google.com/p/javafuse/`

[30] *TurboJPEG: A Java interface for libjpeg-turbo*
`www.libjpeg-turbo.org`

[31] *Numeric performance in C, C# and Java*, Peter Sestoft 2010
`www.itu.dk/people/sestoft/papers/numericperformance.pdf`

[32] *Time Comparing between Java and C++ Software*, Asad Mahmoud Alnaser et al. 2012
`www.scirp.org/journal/PaperDownload.aspx?paperID=21960`

[33] *A Java vs. C++ performance evaluation: a 3D modeling benchmark*, L. Gherardi et al.
`www.best-of-robotics.org/pages/publications/gherardi12java.pdf`

[34] *Release Early, Release Often*, Eric S. Raymond
`www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s04.html`

[35] *The Dependency Inversion Principle*
`www.objectmentor.com/resources/articles/dip.pdf`

[36] *Attacks on Steganographic Systems: Breaking the Steganographic Utilities EzStego, Jsteg, Steganos, and S-Tools and Some Lessons Learned*, Andreas Westfeld and Andreas Pfitzmann 2000

[37] *The Linear Congruential Method*, Donald Knuth, The Art of Computer Programming, Volume 2, p. 9

[38] *Random number generators*, T E. Hull A. R. Dobell 1962, SIAM Review, Volume 4, Issue 3

[39] *PKCS #5: Password-Based Cryptography Standard*, RSA Laboratories.

[40] *The Whirlpool Secure Hash Function*, Williams Stallings 2006
`www.seas.gwu.edu/~poorvi/Classes/CS381_2007/Whirlpool.pdf`

[41] *Random versus Encrypted Data*, A. G. Basile 2008
`opensource.dyc.edu/sites/default/files/random-vs-encrypted.pdf`

[42] *Ent: A Pseudorandom Number Sequence Test Program*, John Walker 2008
`www.fourmilab.ch/random`

[43] *Lexar 8GB USB 3.0 JumpDrive*, Ebuyer 2015
`www.ebuyer.com/399229-lexar-8gb-jumpdrive`

A || Details of the AVI file format

A.1 Detailed AVI form

Listing A.1 shows an expanded form of the `AVI` structure.

```
RIFF (
  'AVI '
  LIST (
    'hdrl'
    'avih'(<Main AVI Header>)
    LIST (
      'strl'
      'strh'(<Stream header>)
      'strf'(<Stream format>)
      [ 'strd'(<Additional header data>) ]
      [ 'strn'(<Stream name>) ]
      ...
    )
    ...
  )
  LIST (
    'movi'
    {SubChunk |
      LIST (
        'rec '
        SubChunk1
        SubChunk2
        ...
        )
      ...
    }
    ...
  )
  ['idx1' (<AVI Index>) ]
)
```

Listing A.1: Detailed `AVI RIFF` form

A.2 The AVI and Bitmapinfo headers

Listing A.2 shows the definition of the main `AVI` header.

```
typedef struct _avimainheader {
  char      fcc[4];
  int32_t   cb;
  int32_t   dwMicroSecPerFrame;
  int32_t   dwMaxBytesPerSec;
  int32_t   dwPaddingGranularity;
  int32_t   dwFlags;
  int32_t   dwTotalFrames;
  int32_t   dwInitialFrames;
  int32_t   dwStreams;
  int32_t   dwSuggestedBufferSize;
  int32_t   dwWidth;
  int32_t   dwHeight;
  int32_t   dwReserved[4];
} AVIMAINHEADER;
```

Listing A.2: The `AVIMAINHEADER` structure.

Listing A.3 shows the definition of the `BITMAPINFOHEADER`.

```
typedef struct tagBITMAPINFOHEADER {
  uint32_t   biSize;
  uint32_t   biWidth;
  uint32_t   biHeight;
  uint16_t   biPlanes;
  uint16_t   biBitCount;
  uint32_t   biCompression;
  uint32_t   biSizeImage;
  uint32_t   biXPelsPerMeter;
  uint32_t   biYPelsPerMeter;
  uint32_t   biClrUsed;
  uint32_t   biClrImportant;
} BITMAPINFOHEADER;
```

Listing A.3: The `BITMAPINFOHEADER` structure.

B || Stegasis example use

B.1 Stegasis usage information

Listing B.1 shows the information displayed to the user when the program is run with no (or incorrect) arguments supplied.

```
   _____ _                        _
  / ____| |                      (_)
 | (___ | |_ ___  __ _  __ _ ___ _ ___
  \___ \| __/ _ \/ _` |/ _` / __| / __|
  ____) | ||  __/ (_| | (_| \__ \ \__ \
 |_____/ \__\___|\__, |\__,_|___/_|___/
                  __/ | v2.1a
                 |___/

Stegasis usage:
  stegasis <command> [-p,-f] -alg=<alg> -pass=<pass> -cap=<capacity>
     <video_path> <mount_point>
--------------------------------------------------------------------------------
Example useage:
  stegasis format -alg=lsbk -pass=password123 -cap=50 /media/video.avi
  stegasis mount -alg=lsbk -pass=password123 /media/video.avi /tmp/test
Commands:
  format  Formats a video for use with stegasis
  mount  Mounts a formatted video to a given mount point
Required Flags:
  -alg  Embedding algorithm to use, see below
  -cap  Percentage of frame to embed within in percent
Optional flags:
  -pass  Passphrase used for encrypting and permuting data
  -pass2 Passphrase used for encrypting and permuting the hidden volume
  -p  Do not flush writes to disk until unmount
  -f  Force the FFmpeg decoder to be used
Embedding Algorithms:
  Uncompressed AVI only:
    lsb:  Least Significant Bit Sequential Embedding
    lsbk:  LSB Sequential Embedding XORd with a psudo random stream
    lsbp:  LSB Permuted Embedding using a seeded LCG
    lsb2:  Combination of lsbk and lsbp
    lsba:  LSB Permuted Embedding encrypted using AES

  Other video formats:
    dctl:  LSB Sequential Embedding within DCT coefficients
    dctp:  LSB Permuted Embedding within DCT coefficients
    dct2:  Combination of dctp and lsbk
    dcta:  LSB Permuted Embedding encrypted with AES
    dct3:  LSB Permuted Embedding encrypted with AES->Twofish->Serpent
```

Listing B.1: `Stegasis` usage information.

B.2 Example use case

Listing B.2 shows an example use of `Stegasis`, formatting and mounting a video with complete command line output.

```
$ stegasis format -alg=lsbp -pass=hunter2 -cap=20 /media/Backup/video.avi
   _____ _                       _
  / ____| |                     (_)
 | (___ | |_ ___  __ _  __ _ ___ _ ___
  \___ \| __/ _ \/ _` |/ _` / __| / __|
  ____) | ||  __/ (_| | (_| \__ \ \__ \
 |_____/ \__\___|\__, |\__,_|___/_|___/
                  __/ | v2.1a
                 |___/
Filesize: 776143108
Totalframes: 280
Width: 1280
Height: 720

Reading AVI chunks...
100% [==================================================]
Finished parsing AVI file

Volume capacity: 19.32MB

Writing back to disc...
100% [==================================================]
Format successful!

$ stegasis mount -alg=lsbp -pass=hunter2 /media/Backup/video.avi /tmp/steg
   _____ _                       _
  / ____| |                     (_)
 | (___ | |_ ___  __ _  __ _ ___ _ ___
  \___ \| __/ _ \/ _` |/ _` / __| / __|
  ____) | ||  __/ (_| | (_| \__ \ \__ \
 |_____/ \__\___|\__, |\__,_|___/_|___/
                  __/ | v2.1a
                 |___/
Filesize: 776143108
Totalframes: 280
Width: 1280
Height: 720

Reading AVI chunks...
100% [==================================================]
Finished parsing AVI file

Header: STEG
Mounting...

[Second Terminal]
$ cd /tmp/steg
/tmp/steg$ cp ~/vid.mp4 .
```

```
[Stegasis Terminal]
Embeding, Frame: 203, Size: 36158, Offset: 0
Compacting header...
100% [==================================================]
Writing back to disc...
100% [==================================================]

[Second Terminal]
/tmp/steg$ ls -lah
total 4.0K
drwxr-xr-x  2 root root    0 Jan  1  1970 .
drwxrwxrwt 10 root root 4.0K Jan 15 21:59 ..
-rwxr-xr-x  1 root root  14M Jan  1  1970 lba2.mp4

[Stegasis Terminal]
<Control-C>
Unmounting...
Compacting header...
100% [==================================================]
Writing back to disc...
100% [==================================================]
Successfully unmounted
```

Listing B.2: `Stegasis` example use.

C || Detailed Code samples

This section contains select detailed code samples.

C.1 Chi-Squared attack

Listing C.1 shows an implementation of the Chi-Squared attack in Python which operates on the PPM image format.

```
from scipy import integrate, special

file_bytes = open(sys.argv[1], 'rb').read()

header = file_bytes[:2]
if header != 'P6':
  print 'File is not a P6 ppm.'
  sys.exit(0)

fp = 3
width = ''
while True:
  b = file_bytes[fp]
  fp += 1
  if b == ' ':
      break
  width += b

height = ''
while True:
  b = file_bytes[fp]
  fp += 1
  if b == ' ':
      break
  height += b

max_pixel_val = ''
while True:
  b = file_bytes[fp]
  fp += 1
  if b == ' ':
      break
  max_pixel_val += b

output = [0]*100

# fp is now on the first pixel red byte
frameStart = fp
for h in range(1, 100):
  fp = frameStart
  totalPixels = math.floor((h/100.0)*int(width)*int(height))
  X = [0]*(128*3) # X[k] = frequency(2k)
  Y = [0]*(128*3) # Y[k] = frequeincy(2k+1)
```

```
    Z = [0.0]*(128*3)

    # Populate the frequency arrays
    end = fp + totalPixels*3
    while fp < end:
        b = ord(file_bytes[fp])
        if b % 2 == 0:
            X[b/2] += 1
        else:
            Y[(b-1)/2] += 1
        fp += 1

    # Calculate theoretically expected frequency
    for i in range(len(Z)):
        Z[i] = (X[i] + Y[i]) / 2.0

    n = 128
    for k in range(127):
        if X[k] + Y[k] <= 4:
            X[k] = 0
            Y[k] = 0
            n -= 1

    X2 = 0.0
    for i in range(128):
        if Z[i] == 0:
            continue
        X2 += ((X[i] - Z[i])**2) / Z[i]

    # Calculate probability of embedding
    p = 1.0 - special.gammainc((n-1)/2.0, X2/2.0)
    output[h] = p

# Print results to stdout
for i in range(1, len(output)):
    print str(i) + " " + str(output[i])
```

Listing C.1: Chi-Squared attack Python implementation.

C.2 Reading and writing to the file system

Listing C.2 shows the final implemented version of the FUSE read function call implementation.

```
int SteganographicFileSystem::read(const char *path, char *buf, size_t size
    , off_t offset, struct fuse_file_info *fi) {
  unordered_map<string, int>::const_iterator file = this->fileSizes.find(
      path);

  if (file == this->fileSizes.end() || offset > file->second)
    return -ENOENT;
  if (size + offset > file->second)
    size = file->second - offset;

  vector<FileChunk> fileChunks = this->fileIndex[path];
  int bytesRead = 0, chunkNum = 0, bytesWritten = 0;

  for (struct FileChunk c : fileChunks) {
    if (bytesRead + c.bytes > offset) {
      break;
    } else {
      bytesRead += c.bytes;
      chunkNum ++;
    }
  }
  while (bytesWritten < size) {
    struct FileChunk chunk = fileChunks.at(chunkNum);
    int chunkOffset = offset - bytesRead;
    int bytesLeftInChunk = chunk.bytes - chunkOffset;
    bytesLeftInChunk = min((int)(size - bytesWritten), bytesLeftInChunk);

    printf("\e[1A");
    printf("\e[0K\rExtracting bytes: %d, offset: %d, frame: %d\n",
        bytesLeftInChunk, chunk.offset, chunk.frame);
    if (chunkOffset == 0) {
      this->extract((int *)&chunk.frame, (int *)&chunk.offset,
          bytesLeftInChunk, buf+bytesWritten);
    } else {
      char *temp = (char *)malloc(chunk.bytes * sizeof(char));
      this->extract((int *)&chunk.frame, (int *)&chunk.offset, chunk.bytes,
          temp);
      memcpy(buf + bytesWritten, temp + chunkOffset, bytesLeftInChunk);
      free(temp);
    }
    bytesWritten += bytesLeftInChunk;
    bytesRead = offset;
    chunkNum ++;
  }
  return size;
};
```

Listing C.2: FUSE read function call implementation.

Listing C.3 shows the final implemented version of the FUSE write function call implementation.

```
int SteganographicFileSystem::write(const char *path, const char *buf,
    size_t size, off_t offset, struct fuse_file_info *fi) {
  int bytesWritten = 0, nextFrame = 0, nextOffset = 0;

  this->decoder->getNextFrameOffset(&nextFrame, &nextOffset);

  struct FileChunk triple;
  triple.frame = nextFrame;
  triple.offset = nextOffset;
  triple.bytes = 0;

  while (bytesWritten < size) {
    this->decoder->getNextFrameOffset(&nextFrame, &nextOffset);

    printf("\e[1A");
    printf("\e[0K\rEmbeding, nextFrame: %d, nextOffset: %d, bytesWritten: %
       d\n", nextFrame, nextOffset, bytesWritten);
    int tmp = this->alg->embed(this->decoder->getFrame(nextFrame), (char *)
       (buf + bytesWritten), size-bytesWritten, nextOffset);
    triple.bytes += tmp;
    bytesWritten += tmp;
  }
  this->fileIndex[path].push_back(triple);

  if (offset == 0) {
    this->fileSizes[path] = size;
  } else {
    this->fileSizes[path] += size;
  }
  this->mux.unlock();
  return size;
};
```

Listing C.3: FUSE write function call implementation.

C.3 Implementing F5

The F5 steganographic algorithm, developed by Andreas Westfeld, embeds data within the LSBs of DCT coefficients within JPEG images. F5 implements matrix encoding to improve embedding efficiency and is not vulnerable to the Chi-Squared and Histogram attacks. The F5 embedding procedure is summarised in Algorithm C.1.

F5 only embeds data within zero valued DCT coefficients. This can lead to a situation known as *shrinkage* which occurs when a coefficient with absolute value 1 is decremented, becoming zero. The extraction processes cannot distinguish between zero valued coefficients skipped and those which were decremented. Therefore when shrinkage occurs, that k-bit message must be embedded again. To estimate the embedding capacity of the carrier medium taking into account possible shrinkage the following formula is used:

Algorithm C.1 The `F5` algorithm.

1: Estimate the embedding capacity of the carrier medium
2: Determine the parameter k from the message size and embedding capacity.
3: Calculate the code word length $n = 2^k - 1$
4: Embed the secret message with $(1, n, k)$ matrix encoding:
5:     Fill a buffer with the LSBs of n nonzero DCT coefficients
6:     Hash this buffer
7:     `XOR` the next k bits of the message with the hash value
8:     If the output i is nonzero, the absolute value of the coefficient at buffer index $i - 1$ is decremented
9:     Test for shrinkage. If so, jump to line 5, else advance the coefficients behind the buffer
10:     If there is still message data, jump to line 5, else return

$$\mathcal{C} = DCT - DCT/64 - DCT_0 - DCT_1 + 0.49 \cdot DCT_1$$

where DCT is the total number of DCT coefficients in the image, DCT_0 the total number of zero valued coefficients and DCT_1 the total number of coefficients with absolute value 1. $-0.51 \cdot DCT_1$ is the estimated loss due to shrinkage.

The embedding rate is defined as:

$$\mathcal{R}(k) = \frac{k}{2^k - 1}$$

and is used to compute the optimal value for k. From the estimated capacity $\mathcal{C}$ and the message length m, the optimal embedding rate is calculated as $r = \dfrac{m}{\mathcal{C}}$ we then seek the maximum integer $k \geq 1$ such that:

$$\mathcal{R}(k) > r$$

The hash function used to hash the LSBs of the nonzero DCT coefficients is the bitwise `XOR` of each element:

$$f(\mathbf{a}) = \bigoplus_{i=1}^{n} a_i \cdot i$$

The index of the coefficient to modify (if any) is calculated by `XOR`ing k bits of the message with the hash of the coefficient buffer as follows:

$$i = M_k \oplus f(\text{buffer})$$

If $i \neq 0$ then the $(i-1)$th buffer elements coefficients absolute value is decremented by 1.

Extraction follows a similar procedure in which blocks of n[26] nonzero valued DCT coefficients are hashed to give k message bits. These need to be collected together and return as a byte array.

[26] There are some fiddly issues here which are not mentioned within the original paper on `F5` since the extraction procedure must know the value of k in advance along with the number of embedded bits

Listing C.4 shows a cut down version of the final implementation.

```
virtual int embed(Frame *c, char *data, int reqByteCount, int offset) {
  this->crypt->encrypt(data, reqByteCount);
  JBLOCKARRAY frame;
  int row, block, co;
  // Estimate the embedding capacity
  int totalCoefficients = this->dec->getFrameHeight() * this->dec->
      getFrameWidth() * 64;
  int zeroCoefficients = 0, oneCoefficients = 0;
  for (int i = 0; i < totalCoefficients; i ++) {
    this->getCoef(lcg.map[i], &row, &block, &co);
    frame = (JBLOCKARRAY)c->getFrameData(row, 1);
    if (frame[0][block][co] == 0) zeroCoefficients ++;
    if (frame[0][block][co] == 1 || frame[0][block][co] == -1)
        oneCoefficients ++;
  }
  // In bits
  int embeddingCapacity = totalCoefficients - totalCoefficients/64 -
      zeroCoefficients - oneCoefficients - 0.49*oneCoefficients;
  // Force to be a multiple of 8
  embeddingCapacity -= embeddingCapacity % 8;
  if (embeddingCapacity < 8) {
    // Not point trying to embed anything in this frame
    int currentFrame, currentFrameOffset;
    this->crypt->decrypt(data, reqByteCount);
    return 0;
  }
  int bitsToEmbed = min(reqByteCount * 8, (int)(embeddingCapacity * (this->
      dec->getCapacity()/100.0)));
  double embeddingRate = (double)bitsToEmbed / (double)embeddingCapacity;
  char k = 1;
  while (true) {
    double rate = (double)k / (pow(2, k) - 1);
    if (rate < embeddingRate) break;
      k ++;
  }
  k --;
  int codeWordLength = pow(2, k) - 1;
  int bitsEmbedded = 0;
  while (bitsEmbedded < bitsToEmbed) {
    int oldOffset = offset;
    int *coefficients = this->getNextCoefficientBlock(c, &offset,
        codeWordLength);
    if (coefficients == NULL) break;
    int hashOfCoefficients = this->hash(coefficients, codeWordLength);
    int dataBlock = this->getNextDataBlock(data, reqByteCount, k,
        bitsEmbedded);
    int index = hashOfCoefficients ^ dataBlock;
    if (index == 0) {
```

within the image. My solution for this was to embed the value of k and the number of embedded bits within the second `JPEG` component using the basic `JSteg` algorithm. The `F5` extraction producer can then access this information.

```
      // Don't need to do anything
      bitsEmbedded += k;
      continue;
    } else {
      // Need to decrement coefficent at index
      index --;
      if (this->decCo(c, oldOffset, codeWordLength, index) == 0) {
        // Shrinkage occured
        offset = oldOffset;
        continue;
      } else {
        bitsEmbedded += k;
        continue;
      }
    }
    free(coefficients);
  }

  return bitsEmbedded / 8;
};
```

Listing C.4: `F5` embedding implementation (`steg/f5.cc:86`)).

D || Testing

D.1 Unit Testing

Unit tests...

D.2 Integration Testing

As briefly mentioned in the implementation section, the integration test suite makes use of a number of test archives. These archives are copied into the volume presented by `Stegasis` and extracted. The resulting file system is then traversed and the contents of files checked to make sure they match their original content. This process is automated using several bash scripts as shown in Listing D.1.

```
#!/bin/bash

function cleanExit() {
  (kill $stegasis_pid)
  exit
}

# Run stegasis
(stegasis format --alg=lsba --pass=test $1)
(stegasis mount --alg=lsba --pass=test $1 /tmp/test) &
stegasis_pid=$!
# Wait for the video to mount
sleep 5

# Copy and extract the archives
cp *.tar /tmp/test
for f in *.tar; do tar xf $f -C /tmp/test; done
rm /tmp/test/*.tar

expected_files="dirfile2.txt  file1.txt  file2.txt  testdir"
files=$(ls -C /tmp/test)
if [ "$files" != "$expected_files" ]; then
  echo "ls returned incorrect file list"
  echo "$files"
  cleanExit
fi

expected_file_1="This is a test file."
expected_file_2="This is a different test file."
file1=$(cat /tmp/test/file1.txt)
file2=$(cat /tmp/test/file2.txt)
if [[ "$file1" != "$expected_file_1" || "$file2" != "$expected_file_2" ]];
    then
  echo "Contents of file(s) incorrect"
  echo "$file1"
  echo "$file2"
  cleanExit
```

```
fi

expected_files_sub="dirfile1.txt"
files_sub=$(ls -C /tmp/test/testdir)
if [ "$files_sub" != "$expected_files_sub" ]; then
  echo "ls returned incorrect file list for sub directory"
  echo "$files_sub"
  cleanExit
fi

expected_file_1_sub="I am in a directory."
file1_sub=$(cat /tmp/test/testdir/dirfile1.txt)
if [[ "$file1_sub" != "$expected_file_1_sub" ]]; then
  echo "Contents of sub directory file(s) incorrect"
  echo "$file1_sub"
  cleanExit
fi


(kill $stegasis_pid)
sleep 3

echo -e "\nAll tests passed :)\n"
exit
```

Listing D.1: The simple integration test suite (`test/simple_integration_tests.sh`).

Other similar tests exist which test different parts of the file system functionality. For example the moving of files using `mv` and copying large amounts of data using `cp`.

E || User study results

Listing E.1 shows the the raw data collected from the user study.

```
Raw data: 0100000110111100100111000000000000010110110000001001011101101011110
    0110101101111011110000001011111001001110111001011100001011000111001110O
```

Listing E.1: Data from the user study.

F || Original Project Proposal

Computer Science Part II Project Proposal

STEGANOGRAPHIC FILE SYSTEMS WITHIN VIDEO FILES

Scott Williams, Christ's College
Originators: Scott Williams

April 9, 2015

PROJECT SUPERVISOR: Daniel Thomas

DIRECTOR OF STUDIES: Professor Ian Leslie

PROJECT OVERSEERS: Professor Peter Robinson, Dr Robert Watson

Introduction and Description of the Work

Steganography is the art of hiding messages within inconspicuous objects - a form of covert communication. Whereas cryptography protects only the content of a message, steganography attempts to conceal the fact that the message even exists. Steganography is particularly useful in countries where encryption is illegal or not suitable, e.g. within the UK, where encryption keys can be forced to be handed over.

There exist many freely available programs which offer message hiding functionality within digital media. However, the majority of these programs operate on single image files and therefore impose a hard limit on the size of message you can embed[27]. Many programs also constrain the type of message you can embed to be a simple text string. Video files on the other hand can be several gigabytes in size without arousing suspicion[28] providing an ideal container for multiple (possibly large) sensitive files. A file system interface would enable users to hide any number of files of any type - just by copying / creating files within the mounted volume. For these reasons, the proposed project focuses on steganographic file systems within video files.

I propose to develop an application which allows a file system to be embedded within a user provided video file. The application will also enable mounting and unmounting of video files with contained file systems. As part of the project I intend to explore a number of steganographic embedding algorithms all of which will be selectable within the final application.

An example use of the final product (henceforward referred to as `Stegasis`):

```
# Prepare an existing video file
$ stegasis format -alg=lsb video.avi

# Using stegasis mount we can directly mount the video file
$ stegasis mount video.avi /mnt/volume

# Create a file inside the file system
$ echo "test" > /mnt/volume/test.txt
# Unmount the file system
$ stegasis umount /mnt/volume
```

After doing some initial research on the topic of steganographic file systems, it seems a suitable approach will be to develop a FileSystem in Userspace using the FUSE package. A similar approach was taken within a paper in which a file system was embedded within multiple JPEG images. For the purposes of this project I'll be focusing on uncompressed raw `AVI` video files.

[27]JPEG images for example are typically only a few megabytes in size - limiting the size of files you can possibly embed.

[28]Raw uncompressed `AVI` files are roughly 2GB per minute of footage.

I propose a staged approach to the project where each stage implements an increasingly secure scheme of embedding the file system, for example starting with naïve least significant bit embedding, showing how this can be broken using statistical analysis and then moving on to more advanced techniques (each method selectable via the –alg flag). The main product of this project - `Stegasis` - as shown above will be a user facing application, enabling all versions of the algorithms described throughout the stages of the project to be run on user provided video files. A number of programs to analyse and break insecure schemes proposed early on during the project could also be produced. This project would tie in nicely with the Part II courses Information Theory and Coding, Digital Signal Processing, and possibly Security II.

Resources Required

I will be using the `C++` programming language to develop `Stegasis` of which I have a good amount of experience with. The virtual File System aspect will be implemented using the `FUSE` package. A scripting language such as `Python` or `MATLAB` may also be used to develop some of the steganalysis tools. Raw `AVI` Video files for testing purposes can be created using `VirtualDub`'s video conversion tools.

I intend to implement the project on my own desktop computer (running Ubuntu 14.04.1 as well as Windows 7) due to convenience and accessibility. However, there is no reason why development could not happen on the PWF machines, should this be needed. Backups will be taken at regular intervals and `Git`, a revision control system will be used (in conjunction with `GitHub`) to preserve multiple versions of the project stored both locally and in an offsite location.

Starting Point

Steganography shares a number of concepts with Cryptography for which an introductory course (Security I) was given last year. I have read the introductory chapters of Steganography in Digital Media by Jessica Fridrich, a number of generic steganography papers and also a few papers specific to Steganographic File Systems.

I have implemented a simple “hello world” `FUSE` virtual file system in `C++` to prove the package works as I would expect.

Substance and Structure of the Project

The project will consist of the following sections:

1. Research and investigation into the theoretical aspects of steganography, identifying appropriate embedding algorithms and steganalysis techniques. Investigation into

developing a virtual file system and the `AVI` video format.

2. Design and implementation of `Stegasis` providing a variety of steganographic embedding algorithms, allowing raw `AVI` files to be formatted, and the mounted file system to be written to and read from. This section will follow an iterative process wherein each iteration will propose an increasingly secure embedding algorithm and an attempt to develop a suitable steganalysis technique to break it.

3. Evaluation of `Stegasis` will be based on the following criteria:
   - Correctness: `Stegasis` correctly formats and mounts a provided video file presenting the file system as a logical volume. Files written to the volume should persist between unmounts and subsequent mounts of the same unmodified video file.
   - Usability: The `Stegasis` command line tool should be simple and intuative providing useage details for its functionality and helpful error messages.
   - Performace: The steganographic embedding process should have no noticeable impact on the file system performance i.e. writes to files should not be perceivably slower than a standard HDD[29].

Success Criteria

For the project to be considered a success, `Stegasis` should provide the following functionality:

- `Stegasis` should offer a number of steganographic embedding algorithms.
- Given a standard raw `AVI` video file, `Stegasis` should format[30] the video such that it can be mounted.
- Given a formatted video file, `Stegasis` should be able to mount the video and present a virtual file system at a given mount point.
- Standard file system operations including listing files, reading a file, writing to a file and deleting a file should be supported within the virtual file system.
- The above described functionality of `Stegasis` should operate without noticeable visual impact on the video content.

[29]For example, it takes roughly 2ms to read a 1MB file from a HDD.
[30]Format in this case is referring to writing some meta data to the video e.g. which embedding algorithm is being used.

Extensions

If there is sufficient time, the following extensions may be attempted:

- Directory Structure: `Stegasis` as described only permits files to be created within the root of the virtual file system. It would be beneficial to allow users to create folder structures as you would expect from a standard file system.
- Audio Usage: `Stegasis` as described only makes use of video image frames to embed the file system. However, a substantial part of an `AVI` file may be the audio data. It would be useful to make use of the audio data to increase the steganographic capacity of the proposed embedding algorithms.
- Video Formats: Unfortunately, raw `AVI` video is (very) uncommon compared with modern compressed video formats such as `H.264 (mp4)` it would be very beneficial for `Stegasis` to operate on a variety of video formats, rather than just raw `AVI`. However, modern video formats are intricate and complex so this may well be outside the scope of this project.
- Evaluation of video artifacts: Quantification of "noticeable visual impact on video content" via a set of human trials possibly achievable by crowd-sourcing through an online website.

Timetable

Michaelmas Term

- ***24th October - 6th November (weeks 3-4):*** Research on the theoretical background of steganography including reading relevant sections of textbooks and academic papers.
- ***7th November - 20th November (weeks 5-6):*** Investigation of appropriate steganographic embedding algorithms suitable for video files. Investigation of the `AVI` video format and the `FUSE` package.
- ***21st November - 4th December (weeks 7-8):*** Implementation of `Stegasis` only offering the simple LSB embedding algorithm and file system functionality.

Winter Vacation

- ***5th December - 18th December:*** Implementation of more advanced steganographic embedding techniques and integration of these into `Stegasis`. Development of steganalysis tools to break proposed embedding schemes.
- ***19th December - 1st January:*** Christmas holiday.

- ***2nd January - 16th January:*** Continuing work on more advanced steganographic embedding and steganalysis techniques.

Lent Term

- ***16th January - 22nd January (week 1):*** Polishing of `Stegasis` and source code (not yet including extension work) - core project should be finished at this point. Write progress report and prepare for the progress presentation.
- ***23rd January - 12th February (weeks 2-4):*** Evaluation of the core project. Identification and implementation of promising extension tasks.
- ***13th February - 26th March (weeks 5-10):*** Work on the dissertation write-up, completing a draft for submission to my supervisor.
- ***27th March - 23rd April (Easter Vacation):*** Revision of dissertation addressing supervisors comments. Dissertation should be ready to submit by 23rd April.