Scott Williams

Computer Science Part II Project Dissertation

STEGANOGRAPHIC FILE SYSTEMS WITHIN VIDEO FILES

Christ's College
University of Cambridge

January 7, 2015

[First Draft]

Performa

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:	12,000
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.

Work Completed

A complete software package has been developed enabling the embedding of arbitrary files within many video formats (including MP4 and AVI) via a file system interface. A total of 9 steganographic embedding algorithms are supported, along with encryption and plausible deniability functionality. Most common file system operations work as expected within the mounted volume and the embedding process can operate without any perceivable impact on video quality. Performance of the system is adequate for general use allowing high definition media content to be played directly out of the embedded volume.

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 innocuous objects. Whereas cryptography seeks to protect only the content of information, steganography attempts to conceal the fact that the information even exists. This allows steganographic methods to be utilised in countries where encryption is illegal for example, or within the UK where keys for identified encrypted data can be forced to be handed over.

In this project I design and implement a practical steganographic software application - `Stegasis` - which enables users to embed arbitrary files within videos via a file system interface. `Stegasis` can operate with no perceivable impact on video quality and can achieve embedding capacities of upto 200% of the video size. A wide range of video formats are supported[1] along with several steganographic embedding algorithms. Standard encryption algorithms can be used to further protect embedded data and plausible deniability functionality protects users even when the presence of embedded data has been confirmed.

Steganogaphic methods operating on video have had comparatively little attention compared to images and audio. As such, there are few programs currently available which allow data to be steganographically hidden within video. `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`[2], 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 many recent global surveillance disclosures show that using certain technologies related to privacy can get you "flagged" by authorities such as the `NSA` - it is no longer the case that simply encrypting data is enough to keep the owner safe.

[1] Including many modern video formats such as `MP4`, `MKV`, `FLV` and `AVI`.

[2] A successful cryptographic program providing on-the-fly encryption and full disc encryption.

2 ‖ Preparation

2.1 Background

In this section steganographic background material, definitions and concepts are introduced. A number of related technologies to the project are also discussed.

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.

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. 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 an active warden may replace words with synonyms or reorder 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 with only 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[3].

2.1.1 Steganographic Concepts

A steganographic system consists of a number of individual components:

- A *Cover object* is the original object that the message will be embedded within.
- A *Message* is an arbitrary length sequence of symbols. For this project we consider a message $\mathcal{M} \in \{0,1\}^n$ that is, a binary string.
- 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.

[3] This unfortunately means utilising video sharing websites such as YouTube and Facebook for distribution is not possible due to them performing compression upon video upload.

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[4], is then formally defined as a pair of embedding and extracting functions Emb and Ext,

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

satisfying,

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

Definition 2.2. Embedding Capacity

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

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

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

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 $\boldsymbol{c} \in \mathcal{C}$ can be expressed as,

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

Note that for certain embedding algorithms, the embedding capacity can depend on both the input data and the cover object[5]. However, in some cases the following expression is also valid,

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

[4]This is specifically steganography by cover modification.

[5]Many algorithms operating on `JPEG` images for example will not embedded 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,

$$\text{LSB}(x) = x \bmod 2$$

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* $LSBP(\boldsymbol{c}, q)$ *which has resolution equal to that of image* $\boldsymbol{c}$ *and with pixel values* $LSBP(\boldsymbol{c}, q)(x, y)$ *given by,*

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

2.1.2 Steganalysis

Steganalysis is the study of detecting messages embedded using steganographic techniques; this is analogous to cryptanalysis applied to cryptography. 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 be likely 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 only (initially) looks at raw uncompressed `AVI` files. Furthermore, only `AVI` version 1.0[6] files are investigated and therefore supported natively[7] by `Stegasis`. Unfortunately, uncompressed `AVI` is today, a very uncommon video format. This is likely due to its relatively huge file sizes when compared to a modern compressed format such as `MP4 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. 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 and one optional chunk. The first sub-list is the file header containing metadata about the video (for example framerate, width and height). The second sub-list contains the actual audio/video data and the optional chunk indexes the offsets of the data chunks within the file.

We therefore have an `AVI` file laid out as follows, see the appendix for a more detailed expanded form.

```
RIFF ('AVI␣'
       LIST ('hdrl' ... )
       LIST ('movi' ... )
       ['idx1' (<AVI Index>)]
      )
```

Listing 2.1: `AVI RIFF` form

With a `RIFF` chunk being defined as follows:

```
struct CHUNK {
  char fourCC[4],
  int ckSize,
  char ckData[ckSize] // contains headers or video/audio data
};
```

Listing 2.2: `RIFF` chunk

And a `RIFF` list defined as:

```
struct LIST {
  char listCC[4], // Will always be the literal 'LIST'
  int listSize,
  char listType[4],
  char listData[listSize]
};
```

Listing 2.3: `RIFF` list

[6] Not including the Open-DML extension (version 1.02).

[7] All other video formats (including compressed `AVI`) are supported via the use of `FFmpeg`, as described in section 3.6.1.

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 is identified by a FourCC tag consisting of a two-digit stream number followed by a two-character code listed in table 2.1.

Two-character code	Description
db	Uncompressed video frame
dc	Compressed video frame
pc	Palette change
wb	Audio data

Table 2.1: AVI stream types

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 euqal to 'DIB ', that is, the AVI contains 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[8].

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

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

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

2.1.4 JPEG compression

The JPEG file format will prove useful when developing a universal steganographic technique operating across many video formats, see section 3.6.1. Steganography within JPEGs has had a comparatively large amount of attention from the research community, most likely due to their popularity and the fact that virtually every camera will produce images in the JPEG format. As such, there exists a fair number of well documented steganographic embedding algorithms for JPEG.

The JPEG compression process consists of 5 main procedures:

[8]This can be inverted via the use of an option within the BITMAPINFOHEADER.

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.

Conveniently, the Independent JPEG Group provide the `libjpeg` C library which 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 -1024 to 1023. A `JPEG` will usually have 3 components corresponding to the *Luminance and Chrominance* colour model (`YCbCr`) with 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, steganographic embedding will usually not occur within the luminance `JPEG` component.

It is worth noting that the `JPEG` decompression and compression processes are computationally expensive. This is especially important when dealing with video since the average 3 minute music video, for example, consists of 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 accommodate this. Also worth noting is that although `JPEG` files are small on disk, they're not once decompressed into RAM. It will not be possible to hold all 4,500 decompressed `JPEG` frames of the average music video in RAM, which is unfortunate again for performance reasons.

2.1.5 FFmpeg

`FFmpeg` is an open source, multimedia framework. 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.

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 other than uncompressed `AVI`. I could continue to investigate more video formats and develop codecs 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`, using `FFmpeg`, prior to the embedding process. However, this doesn't solve the problems of the huge file sizes and

uncommonality of the format. Note that it is not possible to covert to uncompressed `AVI`, perform the embedding and then convert back to the original provided format since the conversion process will be lossy, damaging the embedded data.

A novel solution to this problem is posed in section 3.6.1 and makes use of `FFmpeg` for the video conversion.

2.1.6 Developing a file system

A file system can either operate within *kernel* or *user space*. It was decided at the project proposal stage to develop the file system component for `Stegasis` in user space using the `FUSE` (Filesystem in Userspace) library for a number of reasons. Firstly, 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. Secondly, 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. This is due to the `FUSE` kernel module having to act as a proxy between the system call and the user space code. This is explained below.

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 compete 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. Some of the important operations are discussed below.

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

Listing 2.4: `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`.

The `write` function is similar:

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

Listing 2.5: `FUSE` write operation.

It requests that `size` bytes from the buffer `buf` should be written to the file `path` starting at offset `offset`.

Figure 2.1: Path of a file system call in the hello world example.

Similar functions exist for all of the standard file system operations, see appendix section ??? for a full list.

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[9] 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[10] and none of them provide any sort of file system interface.

[9] 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)

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

2.2.1 StegoStick

`StegoStick` 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 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[11]. 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` 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 echos this to the user.

2.2.3 TCSteg

`TCSteg` 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.

[11] Listed in the source as "44 byte header + 54 bytes of extension space".

2.2.4 StegoVideo

`StegoVideo` is a Virtual Dub filter[12] 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` claims to protect the embedded data via the use of a passkey (a 5 digit number), although as with `StegoMagic`, this is not provided by the user and is instead generated and presented to the user to make a note of.

2.2.5 OpenPuff

`OpenPuff` 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%[13] 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` is a cross platform steganograhpic program developed by James Ridgway. It allows users to embed a file within `MP4` (`H264`) files via the modification of motion vectors. The input file is encrypted using `AES` with a user provided pass phrase. Need to test this in Linux.

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 providing the user the opportunity to embed an arbitrary number of files within a video via a file system interface and providing high capacity steganographic embedding algorithms offering capacities in excess of 100%[14]

[12] Which is also available in a stand alone executable form.
[13] 2,600 bytes within a 60 MB video.
[14] This specifically means a file of size n bytes can have more than n hidden bytes residing inside it.

of the cover object's size[15].

2.3 Requirements Analysis

After reviewing the necessary background material and investigating current available solutions to the problem of steganography within video, the following collection of requirements were produced. For the project to be considered a success, at least all of the core requirements should be fulfilled.

2.3.1 Core Requirements

`Stegasis` should:

- Allow users to embed data within video files:
  - Several steganographic embedding algorithms should be available.
  - Each embedding algorithm, $\mathcal{A}$, should satisfy correctness. That is,
    $$\forall\ \mathbf{c}, \mathbf{k}, \mathbf{m}.\ Ext_{\mathcal{A}}(Emb_{\mathcal{A}}(\mathbf{c}, \mathbf{k}, \mathbf{m}), \mathbf{k}) = \mathbf{m}.$$
  - Steganalysis tools should be developed to test the security of the proposed embedding algorithms.
  - An optional user provided password should encrypt data prior to embedding.
  - A capacity option should allow users to specify the percentage of each video frame to embed within.
- Provide a file system interface:
  - The presented logical volume should reside at a user provided mount point.
  - Data written to the file system should be embedded on the fly within the chosen video file.
  - Data accessed from the file system should be retrieved on the fly from within the video.
  - 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.
- Support raw uncompressed `AVI` video:
  - Uncompressed `AVIs` should be natively parsed allowing access to individual pixel data.
- Provide performance adequate for normal use:
  - Full HD video content should be playable directly from within the presented file system.

[15] This is very much a trade off - capacities larger than the file size will come at the sacrifice of steganogrphic security. However, this decision is presented to the user rather than decided by the program itself.

2.3.2 Possible Extensions

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

`Stegasis` should:

- Support a wide range of video formats:
  - Specifically including the popular video format `MP4`.
- Allow directory operations within the file system:
  - Creating directories using the `mkdir` command should work as expected, as should using the `mv` and `rm` commands.
  - Organising files within directories should also work as expected.
- Embed also within audio data:
  - Data should also be embedded within the (possible) audio stream of the video, therefore increasing the embedding capacity.
- Provide plausible deniability:
  - A second file system should be (optionally) embedded within the video, mountable with a second passphrase.
  - The presence of the second, hidden file system should not be detectable.
- Be evaluated for perceivable video impact using a web application:
  - The web application should evaluate the claim "Embedding has 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 languages and tools can be identified.

It is first noted that `Stegasis` (as developed for this project) will only function on the `Linux` operating system. That is, there is no requirement for `Stegasis` to be cross platform.

The file system is an important aspect of `Stegasis` and so it is initially decided which approach to take in developing it as this will influence the later choice of an appropriate programming language. As described within section 2.1.6, it was decided to use the `FUSE` library to develop the file system component in user space.

We now address the choice of primary programming language for the development of `Stegasis`.

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. The identified library for implementing the file system aspect - FUSE - is natively a C library (as is the libjpeg library, and libraries provided by FFmpeg). Although wrappers for other languages (including Java) do exist, 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 SteganographicAlgorithm interface of which each embedding algorithm implements. This suggests C++ over C. The final core requirement, performance, also favours C/C++ over Java[16] due to the JVM overheads.

The reasons above 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 website 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.

3 ‖ Implementation

3.1 Introduction

The development of Stegasis consisted of the 5 main stages sectioned 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[17]. The testing section provides an overview of the testing processes applied throughout development.

[16] There have been numerous studies showing that C/C++ code performs better than equivalent Java code.

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

The actual software development process taken differs from that laid out below; each section was not wholly completed before moving onto the next. Instead, an iterative process was taken across all sections, embracing the modern "Launch early, iterate often" methodology. For example, as specified in the project proposal timetable, a simplified version of `Stegasis` was initially produced only offering one simple embedding algorithm and basic file system functionality. This allowed 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. For the sake of readability, I have structured the sections below to group together implementation details for each separate component.

3.2 AVI Decoder

The concept of an `AVI` decoder is first abstracted to that of a generic `Video Decoder` interface[18] 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. It was therefore decided to define the `Video Decoder` to allow access to the video pixel data at a granularity of a single video frame. It would 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 following definition for the `Video Decoder` interface (`NextFrameOffset` will be discussed in section 3.4):

```
class VideoDecoder {
  public:
    virtual Chunk *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 getNextFrameOffset(int *frame, int *offset) = 0;
    virtual void setNextFrameOffset(int frame, int offset) = 0;

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

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

Note that `getFrame` returns a `Chunk` wrapper object, rather than a raw `char` pointer to the frame pixel data, adhering to the *Dependency Inversion* principle. This will be useful

[18]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.

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 `Chunk` 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 `Chunk` interface is therefore defined as follows:

```
class Chunk {
  protected:
    long chunkSize;
  public:
    virtual long getChunkSize() = 0;
    virtual char *getFrameData(int n=0, int c=0) = 0;

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

Listing 3.2: Chunk 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` parssing process can be thought of consisting of two main parts; parsing the video headers and parsing the video chunk data. The following pseudocode illustrates this with the headers being parsed lines 1-12 and the chunks being parsed lines 15-22: See the appendix section B 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 `Chunk` data into the original `AVI` file. This operation is described in algorithm 3.2 below.

The remaining functionality of the parser is mostly trivially returning data from the `aviHeader` structure (for example `aviHeader.height`, `aviHeader.totalFrames` etc.) The only other function of interest is `getFrameSize`, which returns the number of bytes within each frame that can be embedded within. This (incorrectly) assumes that each frame can always have the same number of bytes embedded within it. This assumption is however true for the steganographic algorithms presented in the next section[19], and is implemented as in listing 3.3.

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

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

[19]It is also true for the simple `JPEG` embedding algorithms used within the extension tasks.

Algorithm 3.1 `AVI` parsing process

```
1: f ← open(file_path)
2: riff_header ← readRiffHeader(f)
3: if riff_header.fourCC != RIFF then
4:     print "File is not an AVI file"
5:     Exit
6: avi_header ← readAviHeader(f)
7: bitmap_info_header ← readBitmapInfoHeader(f)
8: if bitmap_info_header.compression != 0 then
9:     print "Stegasis does not natively support compressed AVI files"
10:    print "Rerun using the -f flag"
11:    Exit
12: audio_info_header ← readAudioInfoHeader(f)
13: frame_chunks ← [ ]
14: i ← 0          ▷ File pointer is now positioned at the start of the audio video chunks
15: while i < avi_header.total_frames do
16:    chunk ← readChunk(f)
17:    if chunk.fourCC == 00db then
18:        frame_chunks[i].chunkSize = chunk.chunkSize
19:        frame_chunks[i].frameData = readChunkData(f)
20:        i ++
21:    else
22:        Advance f chunk.chunkSize bytes                ▷ Chunk was not a video chunk
```

Algorithm 3.2 `AVI` write back process

```
1: Seek f to the chunks offset
2: i ← 0
3: while i < avi_header.total_frames do
4:     chunk ← readChunk(f)
5:     if chunk.fourCC == 00db then
6:         if frame_chunks[i].isDirty then
7:             Write frame_chunks[i].frameData to f
8:             frame_chunks[i].dirty = false              ▷ This chunk is no longer dirty
9:         else
10:            Advance f chunk.chunkSize bytes      ▷ Chunk did not need to be written
11:        i ++
12:    else
13:        Advance f chunk.chunkSize bytes                ▷ Chunk was not a video chunk
```

This expression arises from the fact that uncompressed `AVI` uses 24 bits (3 bytes) per pixel value. So since there are $height \cdot width$ pixels within a single frame, we simply multiply this by 3 to get 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.

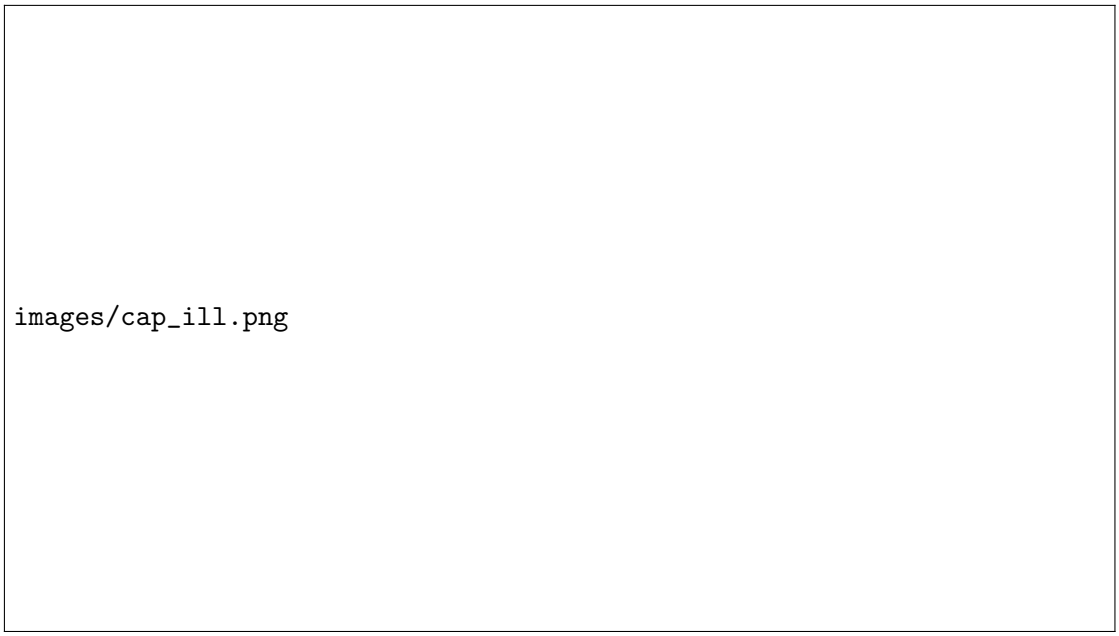

Figure 3.1: Illustration of the capacity parameter.

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

The concept of a generic stegangraphic embedding algorithm is initially developed and implemented as a `Steganographic Algorithm` interface. As defined within the background discussion of steganogrphic concepts, a steganogrphic system consist of a pair of functions providing embedding and extraction functionality. The interface will therefore need to declare two functions `embed` and `extract` which will embed and extract data respectively. The `Video Decoder` interface defined above allows access to video data via individual frame `Chunk` objects. It was therefore decided to define the `embed`

and `extract` functions to operate on `Chunk` objects. The following interface definition was therefore used.

```
class SteganographicAlgorithm {
  protected:
    string password;
    VideoDecoder *dec;
  public:
    virtual void embed(Chunk *c,char *data,int dataBytes,int offset)=0;
    virtual void extract(Chunk *c,char *output,int dataBytes,int offset)=0;
    virtual void getAlgorithmCode(char out[4]) = 0;
};
```

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

The `embed` function should be read as "embed `dataBytes` bytes from `data` into chunk `c` starting at an offset `offset` bytes into the frame". Similarly, the `extract` function should be read as "extract `dataBytes` bytes from chunk `c` starting at an offset `offset` bytes into the frame and put them into `output`". It was decided to place the error handling logic within the file system code. The steganogrphic algorithm implementations therefore do not contain any such logic and instead assume that pointer `data` does indeed point to at least `dataBytes` bytes and so on.

`getAlgorithmCode` simply returns a (unique) 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 message bits producing the stego image. Algorithm 3.3 show pseudocode for the LSB embedding algorithm.

Algorithm 3.3 LSB embedding algorithm

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

The matching extraction algorithm is shown in algorithm 3.4.

Algorithm 3.4 LSB extraction algorithm

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

The actual implementation is relatively short and so is included below in listing 3.5.

```
virtual void embed(Chunk *c, char *data, int dataBytes, int offset) {
  char *frame = c->getFrameData();
  for (int i = 0; i < dataBytes; i ++) {
    for (int j = 7; j >= 0; j --) {
      if ((((1 << j) & data[i]) >> j) == 1) {
        frame[offset++] |= 1;
      } else {
        frame[offset++] &= ~1;
      }
    }
  }
};
```

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

The operation of the sequential LSB embedding algorithm is illustrated in figure 3.2.

Figure 3.2: Illustration of the LSB Embedding algorithm.

Using the algorithm as shown above, 43.2 kB of data is is embedded into a 1280 × 720 resolution video frame (a capacity setting of 100%). The result of this is shown in figure 3.3. The left image is the cover object and the right image is the stego object.

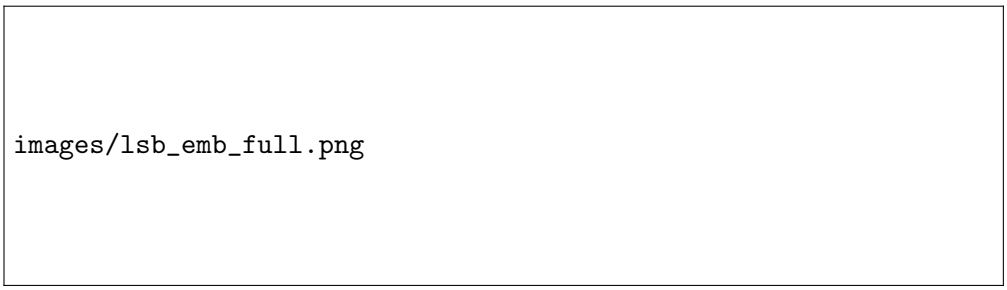

Figure 3.3: Effect of the LSB Embedding algorithm.

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 a closer look at a specific portion of the video frame we can see some small discrepancies between the cover and stego objects - figure 3.4. I am unsure how well these images will be reproduced when printed, but the difference is definitely noticeable within the PDF. Note that without the original cover object for comparison, it would be very hard (if not impossible) 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 another steganalysis attack known as the *Chi-Squared attack* developed by Westfeld and Pfitzmann. To implement the Chi-Squared attack, the concept of *Pair of Values* (PoVs) is first introduced.

One of the results 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.

To implement the attack, the following steps are followed. First, $x_k = \text{frequency}(2k)$ and $y_k = \text{frequency}(2k+1)$ are calculated $\forall\ k$, followed by the expected frequency $z_k = \dfrac{x_k + y_k}{2}$. n is defined to be the number of POVs, that is $|\mathcal{P}ov|$. 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:

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

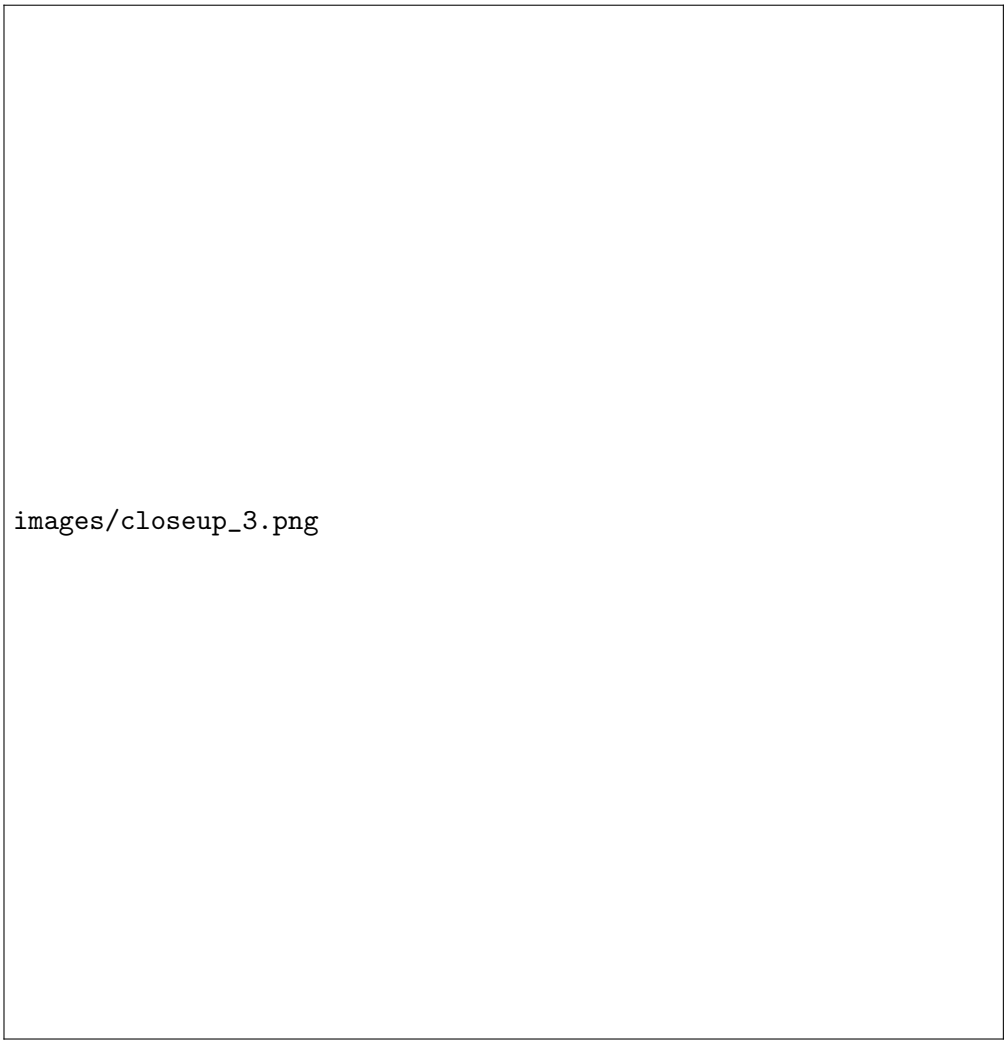

Figure 3.4: Close up effect of the LSB Embedding algorithm.

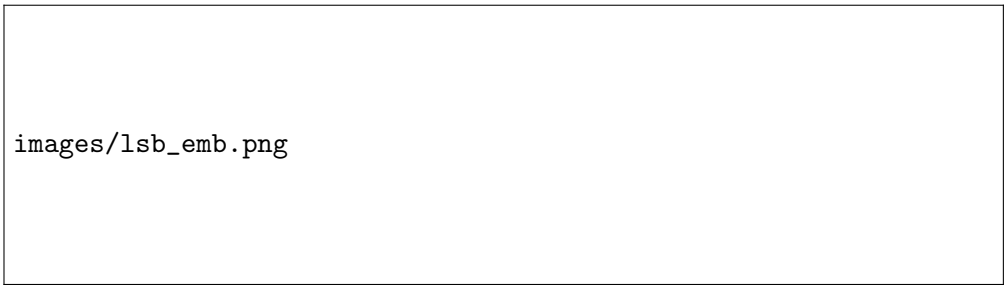

Figure 3.5: LSB plane of the cover and stego object.

See appendix section ?? 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 firth half of the frame.

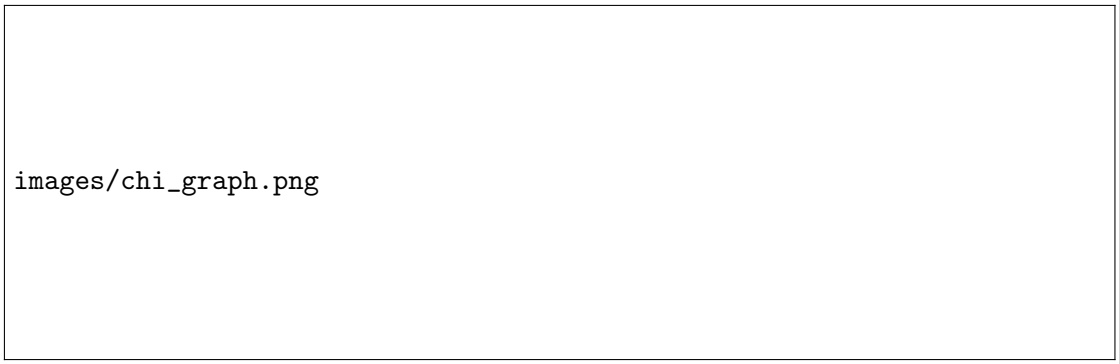

Figure 3.6: Results of the Chi-Squared attack.

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 video frame. A permuted LSB embedding algorithm achieves this and resists the Chi-Squared attack when using a small capacity setting.[20].

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
```

[20]Of course, a permuted embedding using a capacity of 100% is comparatively no different to sequential embedding!

Lines 1 and 2 hide a large amount of complexity involved in actually implementing the permuted LSB 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. The obvious way to produce a pseudorandom permutation of a list of 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, a different approach using a *Linear Congruential Generator* (LCG) was taken.

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 Hull-Dobell Theorem 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 thrown away, therefore producing a full period of size n as required.

The pseudorandom permutation should be dependant on a user provided password. Therefore, the values of a and c are determined using a key derived from the users password. The popular key derivation function `PBKDF2` is used along with the `Whirlpool`

hash function to generate a 128 byte key 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 the popular C++ cryptographic library, `Crypto++`.

This successfully describes an implementation for line 1 of algorithm 3.5. Line 2 unfortunately causes some problems for the LCG due to precisely the reason that it was chosen to be used.

Since the full array of the pseudorandom permutation is not being stored, the LCG does not facilitate direct random access. For example, if the 50th element of the permutation is requested, the 49 elements before it must first be calculated starting from X_0. This had a huge impact on the performance of the algorithm to the point of the file system becoming unusable. Therefore, it was decided to pre-compute a hashmap mapping from frame offsets to the corresponding pseudorandom permutation element, therefore eliminating the need to calculate all elements prior to the requested one. This however now means the entire permutation is being stored in memory - exactly why the original method was rejected! It was decided to remain using the LCG method with only a single global permutation used across all video frames.

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%.

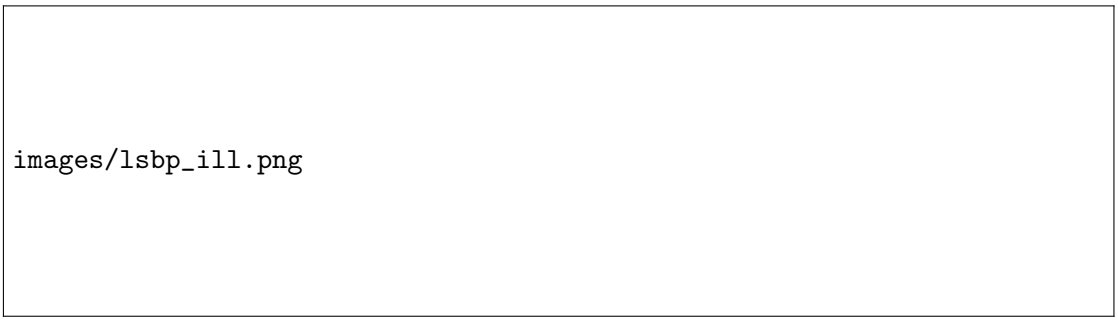

Figure 3.7: Illustration of the permuted LSB algorithm.

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 lot better than some of the those provided by programs investigated in section 2.2, and

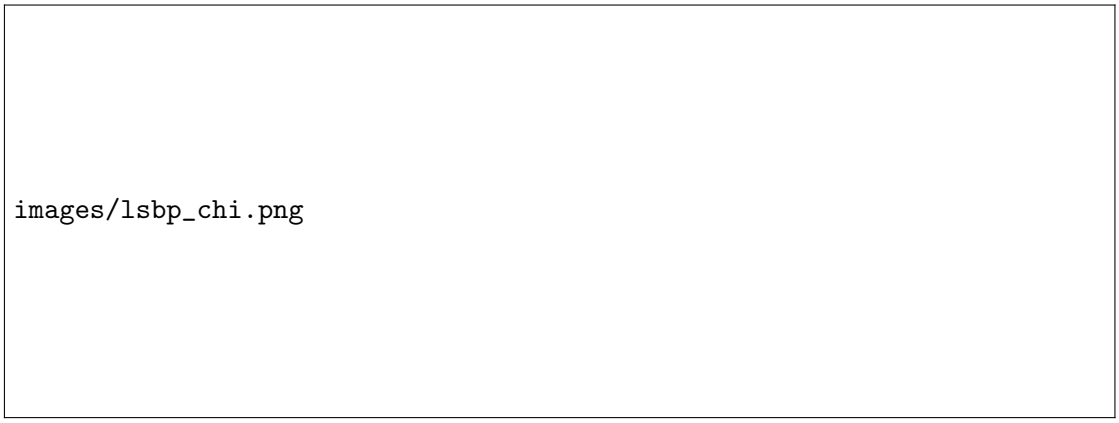

Figure 3.8: Chi-Squared attack on the permuted LSB algorithm.

will be further improved upon during the extension task within section 3.6.1.

3.3.3 Permuted and Encrypted LSB Embedding

To further strengthen the security of `Stegasis`, it was decided to encrypt the data before embedding it within the video. This way, even if the stegosystem is broken, that is the hidden data has been identified and extracted, the data itself will not be compromised. The same user provided password used to permute the data throughout the video frame is used within the encryption process, addressing another concern of some of the investigated programs in section 2.2.

The initial approach to encrypting the embedded data was to simply `XOR` the data stream with a pseudorandom number stream. The `Crypto++` library provided a variety of pseudorandom number generators which could be seeded and used as the pseudorandom number stream. It was chosen to use the `RandomPool` algorithm and to seed it using both the entire 128 byte key derived using the `PBKDF2` algorithm, and the requested embed offset into the video frame. This way, if the same file is written twice to the volume, it will be encrypted to 2 different byte streams[21]. Unfortunately, this will cause some problems when the file system is developed in section 3.4.

Listing 3.6 shows the implementation of the permuted and encrypted LSB embedding algorithm.
More standard encryption algorithms were then added, such as `AES` (256 bit), `TwoFish` and `Serpent`. All of these algorithms are available within the `Crypto++` library which made incorporating them easy. `Stegasis` currently supports standard `AES`, and also `AES` $\rightarrow$ `TwoFish` $\rightarrow$ `Serpent`. That is, the data is first encrypted using `AES`, the result of this is then encrypted with `TwoFish` etc. This method has the advantage that if even 2 of the

[21] Unless the file is embedded within 2 different frames at the same offset.

```
virtual void embed(Chunk *c, char *data, int dataBytes, int offset) {
  char *frame = c->getFrameData();
  CryptoPP::RandomPool pool;
  pool.IncorporateEntropy((const unsigned char *)this->key, 128);
  pool.IncorporateEntropy((const unsigned char *)&offset, 4);

  int frameByte = lcg.map[offset++];
  for (int i = 0; i < dataBytes; i ++) {
    char xord = data[i] ^ pool.GenerateByte();
    for (int j = 7; j >= 0; j --) {
      if ((((1 << j) & xord) >> j) == 1) {
        frame[frameByte] |= 1;
      } else {
        frame[frameByte] &= ~1;
      }
      frameByte = lcg.map[offset++];
    }
  }
};
```

Listing 3.6: Encrypted and permuted embedding (`steg/lsb2_algorithm.cc:41`)

above algorithms are cryptographically broken, the data will still be secure[22]. You do however incur a performance penalty due to the encryption, however the performance of the file system is still good enough to facilitate the playback of HD video content directly out of it.

This concludes the steganogrphic embedding algorithms developed to satisfy the core requirements operating on uncompressed `AVI` video. Although I have shown resistance to some steganalysis techniques, I have no doubt that there exist attacks that would break the implemented stego systems. However, steganogrphic security was not the sole focus of this project and the framework `Stegasis` provides allows easy incorporation of new, more secure embedding algorithms.

3.4 The File system

It was decided early on and stated within the project proposal 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 practical file system and were planned to be implemented: Creating and deleting files, reading and writing to files, listing the files in the file system and renaming (moving) files. The lack of directory functionality means that all embedded files will reside in the root of the file system, also note the lack of permissions support, all files will be given the same permissions of `755` (`RWXRW-RW-`). Changing a file's access and modification times is also not implemented meaning all times default to 0 *Unix Time* (1 January 1970). Omitting these features does have an

[22]This encryption option is also offered by `TrueCrypt`.

impact on the usability of the file system, for example the `make` command makes use of file modification dates to determine if they need recompiling. However, the file system is not being designed for general purpose use, it is merely supposed to be a container for a collection of files[23].

Since the file system functionality was being implemented from scratch, the code written turned out very 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 the 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[24]. Now, additional text is appended to the first file, where should this be embedded? The obvious answer is after the second file, requiring some sort of header to keep track of the file chunks so that the files can be correctly read back.

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. It was decided to use the first frame of the video to store the header and is such named the *File Allocation Frame* or `FAF` for short. 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 header.
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 in the header (that is, the size of the header not including the blue coloured section), allowing the extraction of the remaining header data from the `FAF`. The remaining header data contains information for each of the files within the file system. This includes the file name as a null terminated string, the number of chunks the file is split into and the location and size of each of these chunks. A file chunk is represented as a `Triple` defined in listing 3.7.
Within figure 3.9, a single triple is indicated with square brackets, and is coloured green. A file can contain an arbitrary number of triples each of which can span multiple frames,

[23] However, I would argue that after the directory structure extension task has been implemented, the file system provides enough functionality for most general use cases.
[24] Since it takes 8 bytes of frame to embed 1 byte of data.

Figure 3.9: Header structure overview.

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

Listing 3.7: Triple definition (`fs/stegfs.h:19`)

but note that using an unsigned integer for the struct fields limits individual chunk sizes to (roughly) 4 GB. Also note that only a single frame is used as the `FAF` - it cannot span multiple frames. This limits the total size of the header, imposing a maximum number of files and chunks.

3.4.2 Reading and writing the header

Internally, the header data will be stored in memory as a pair of `unordered_maps`, one mapping from file name to file size and the other from file name to a vector[25] of `Triples`. These are named `fileSizes` and `fileIndex`.

The process of reading the header is split up into 2 main sections, reading the header data and reading the file data. These are both illustrated within Algorithm 3.6.

The video decoder get and set frame offset methods, briefly mentioned in section 3.2, are used to indicate where new files written to this existing file system should be

[25]This is an ordered data structure, the triple contained in vector[0] will contain the file data immediately preceding vector[1].

embedded within the video. The read header process identifies the highest frame and offset and passes this data to the video decoder.

Writing back the header is a similar process explained within algorithm 3.7.

Algorithm 3.6 Reading the file system header.

```
1:  header_frame ← the first video frame
2:  header_sig ← extract 4 characters from header_frame
3:  if header_sig != STEG then
4:      print "Video has not been formatted, or invalid password."
5:      Exit
6:  algorithm_code ← extract 4 characters from header_frame
7:  capacity ← extract byte from header_frame
8:  header_bytes ← extract integer from header_frame
9:  while extracted < header_bytes do
10:     file_name ← extract string from header_frame
11:     number_triples ← extract integer from header_frame
12:     file_size ← 0
13:     for j ← 0 upto number_triples - 1 do
14:         triple ← extract triple from header_frame
15:         fileIndex[file_name].append(triple)
16:         file_size += triple.bytes
17:     fileSizes[file_name] ← file_size
18: max_frame ← largest frame containing data
19: max_offset ← smallest offset within max_frame data is not embedded within
20: video_decoder.setNextFrameOffset(max_frame, max_offset)
```

3.4.3 Compacting the header

During testing, it was noticed that writes to the file system seemed to occur in chunks of `4096` bytes. Due to the above presented design of the file system header, this resulted in relatively large files containing a very large number of chunks, too large to fit into the singe `FAF`. For example, a 5 MB music file would be written into 1221 chunks. Since each chunk consists of 12 bytes (three 4 byte integers), you would need 14.65 kB to just store all of the triples into the header. Unfortunately, this meant the largest file you could embedded within any 720p uncompressed `AVI` (regardless of the length) was 117 MB. This was decided to be an unacceptable limit and so a process to compact the header and remove unnecessary chunks was developed. Figure 3.10 shows a visual representation of this process occurring, green and red lines indicate the start and end of a chunk respectively. The first process compacts sequential chunks of the same file within frames to give single chunks per frame, sequential single frame chunks are then combined in the second process. This allows a file spread across the entire video to be represented by a single chunk.

Algorithm 3.7 Writing the file system header.

```
1: header_frame ← the first video frame
2: header ← byte array
3: header_bytes ← 0
4: offset ← 0
5: for (file_name, triples) in fileIndex do
6:     header[offset] ← file_name
7:     offset += file_name.length() + 1                    ▷ Null terminated
8:     num_triples ← triples.length()
9:     header[offset] ← num_triples
10:    offset += 4
11:    for triple in triple do
12:        header[offset] ← triple
13:        offset += sizeof(triple)
14:    header_bytes += file_name.length() + 1
15:    header_bytes += 4
16:    header_bytes += sizeof(triple)*num_triples
17: embed header_bytes into header_frame
18: embed header into header_frame
```

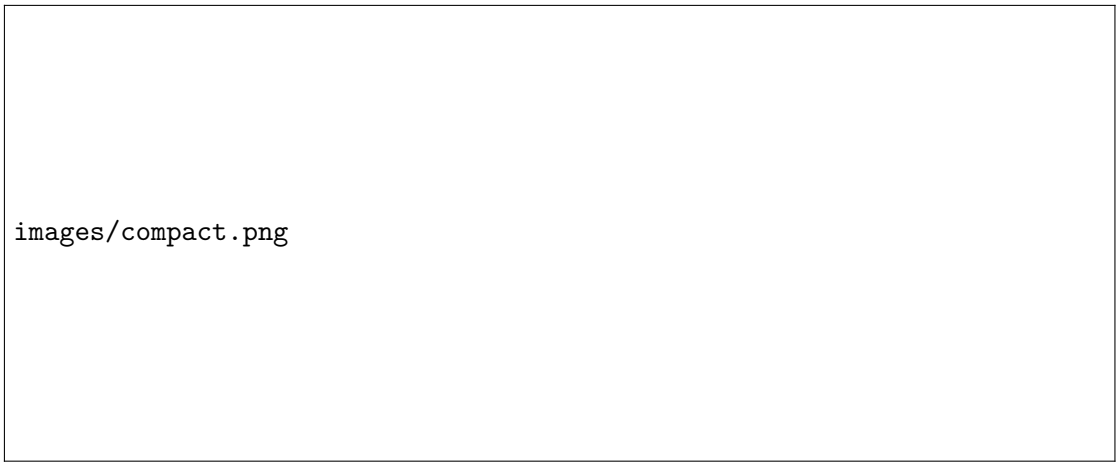

Figure 3.10: Header compaction process.

3.4.4 Writing to the file system

A write to the file system will occur 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 3.8.

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

Listing 3.8: The `create` function call (`fs/stegfs.cc:202`).

The write calls are a bit trickier, recall the `FUSE` write call:

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

Listing 3.9: `FUSE` write operation.

Which requests that `size` bytes from the buffer `buf` should be written to the file `path` starting at offset `offset`. Algorithm 3.8 shows pseudocode for the write function.

Algorithm 3.8 Writing to the file system.

```
1:  bytes_written ← 0
2:  while bytes_written < size do
3:      (next_frame, next_offset) ← decoder.getNextFrameOffset()
4:      frame ← getFrame(next_frame)
5:      triple ← Triple()
6:      triple.frame ← next_frame
7:      triple.offset ← next_offset
8:      bytes_left_in_frame ← Frame Size - next_offset
9:      if size - bytes_written < bytes_left_in_frame then
10:                                          ▷ This frame will finish off this write call
11:         triple.bytes ← size - bytes_written
12:         embed triple.bytes from buf+bytes_written into frame at offset next_offset
13:         decoder.setNextFrameOffset(nextFrame, next_offset + size-bytes_written)
14:         bytes_written += size-bytes_written
15:     else
16:                                  ▷ Write all bytes left in frame and go around again
17:         triple.bytes ← bytes_left_in_frame / 8
18:         embed triple.bytes from buf+bytes_written into frame at offset next_offset
19:         decoder.setNextFrameOffset(nextFrame + 1, 0)
20:         bytes_written += bytes_left_in_frame / 8
21:     fileIndex[path].append(triple)
22: return size
```

The above explanation doesn't address the case where writes occur to a file already existent within the file system, that is, it is being overwritten or extended. This is more complicated since it is beneficial to reuse the previously allocated chunks for the file and is therefore listed in the appendix section ??.

Finally, `flush` is called which will write the header and ask the video decoder to write back to disk. Note the conditional flag *performance*, it is not necessary to flush to disk and by running `Stegasis` with the `-p` flag, this will be skipped.

```
int SteganographicFileSystem::flush(const char *path, struct fuse_file_info
    *fi) {
  if (!this->performance) {
    this->writeHeader();
    this->decoder->writeBack();
  }
  return 0;
};
```

Listing 3.10: `FUSE` flush implementation (`fs/stegfs.cc:537`).

3.4.5 Reading from the file system

The read function call is similar, recall the declaration,

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

Listing 3.11: `FUSE` read operation.

which requests that `size` bytes of the file `path` starting at offset `offset` should be written to the buffer `buf`. Rather than allocating new chunks, 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 3.9 describes the function operation. Note that two cases are needed when extracting data from chunks due to the possibility of data being encrypted. If `read` requests data from the middle of a chunk, the entire chunk is first decrypted, and then the correct data returned from this.

The described algorithm doesn't work for chunks spanning multiple frames as it is rather verbose to list here, a complete version which does is listed in appendix section ??.

3.4.6 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 3.12.

```
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 3.12: `FUSE` readdir implementation (`fs/stegfs.cc:264`).

Algorithm 3.9 Reading from the file system.

```
1: bytes_written ← 0
2: (triple, chunk_offset) ← findTriple(offset)
3: while bytes_written < size do
4:     bytes_left_in_chunk ← triple.bytes - chunk_offset
5:     frame ← getFrame(triple.frame)
6:     if size - bytes_written < bytes_left_in_chunk then
7:         if chunk_offset == 0 then
8:             extract size-bytes_written bytes from frame at offet triple.offset into buf+bytes_written
9:         else
10:            extract entire triple from frame into temp
11:            copy size-bytes_written bytes from temp+chunk_offset into buf+bytes_written
12:        return size
13:    if chunk_offset == 0 then
14:        extract bytes_left_in_chunk bytes from frame at offet triple.offset into buf+bytes_written
15:    else
16:        extract entire triple from frame into temp
17:        copy bytes_left_in_chunk from temp+chunk_offset into buf+bytes_written
18:    bytes_written += bytes_left_in_chunk
19:    chunk_offset ← 0
20:    triple ← getNextTriple()
```

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. It was decided that a command line application would be developed allowing the user to utilise the functionality developed within this project. Listing 3.13 shows the process of formatting and mounting a video.

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

Listing 3.13: Using `Stagasis` to format and mount a video.

At this point, the user can copy files into `/mnt/video` and they will be 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 ?? for some 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[26].

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 in the preparation section, there exist several reasons why it would be beneficial for `Stegasis` to operate on video formats other than uncompressed `AVI`. To accomplish this, rather than develop many steganographic methods 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. This idea is used to allow a universal steganographic method operating on sequences of images to operate across all video formats. Figure 3.11 shows an overview of the approach.

The user specified video is first converted into a sequence of images, one for each frame of the video. Steganographic embedding then occurs within these images before being reassembled back into a video[27]. The `JPEG` file format was chosen for the intermediate frames due to the large number of steganographic algorithms operating on them. `FFmpeg` facilitates the extraction and assembling of the video frames, allowing this method to work with virtually any video format due to its extensive codec library. Note that the extraction of video frames will be a lossy process, leading to small visible `JPEG` compression artifacts when comparing the original video and the extracted frames. This is (initially) not a issue for the proposed technique and is not an indication of steganographic embedding. However, the reassembly process cannot be lossy. The data embedded using the identified embedding algorithms operating on `JPEG` images will not survive recompression. This constraint leads to the following pair of `FFmpeg` commands shown in listing 3.14 and 3.15.

The frame extraction command extracts each frame of the video file using the highest quality setting and writes them as `JPEGs` to `/tmp/stegasis/`. The reassembly command *muxes* the modified video frames and audio together into a single `MKV` file. Note the use of `-codec copy`, this 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

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

[27] The (possible) video audio is extracted separately and then recombined during the assembly process.

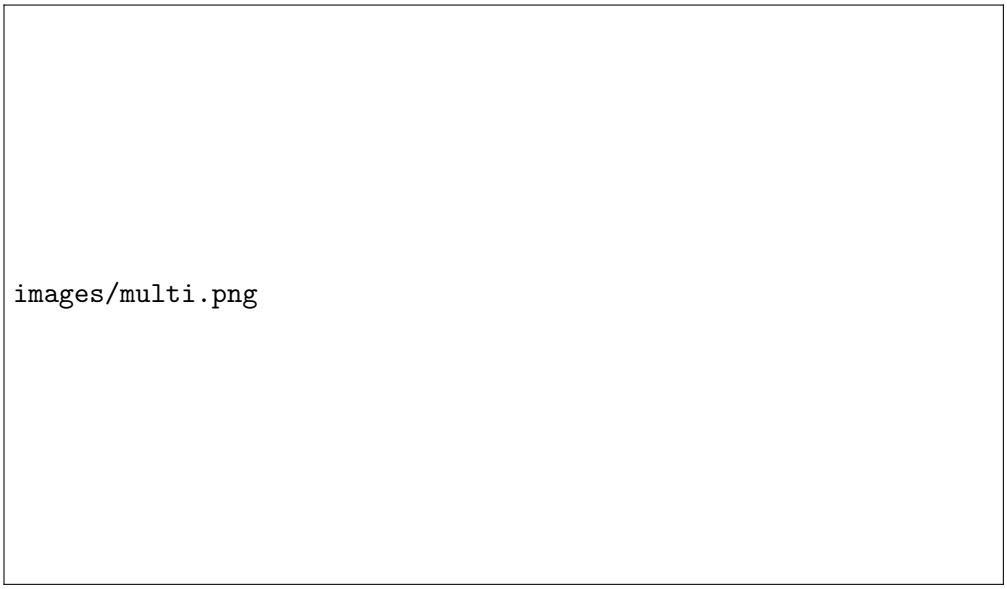

Figure 3.11: Embedding within multiple video formats.

```
ffmpeg -r <video_fps> -i <video_path> -qscale:v 1 -f image2 /tmp/stegasis/
    image-%d.jpg
ffmpeg -i <video_path> /tmp/stegasis/audio.mp3
```

Listing 3.14: `FFmpeg` frame extraction command.

prior to being muxed. These frames were then reextracted from the resulting `MKV` file, their hashes computed and compared.

This method allows `Stegasis` to operate on an arbitrary video file, but not to remount the produced `MKV` file. The above paragraph states that the extraction process being lossy is (initially) not a problem. However, if we extract the frames of a produced `MKV` file using a lossy process, the data embedded within the frames will be lost. To combat this, a third `FFmpeg` command is used when remounting an already formatted `MKV` file, as shown in listing 3.16.

```
ffmpeg -r <video_fps> -i <video_path> -vcodec copy /tmp/stegasis/image-%d.
    jpg
```

Listing 3.16: `FFmpeg` `MKV` `MJPEG` fram extraction command.

```
ffmpeg -r <video_fps> -i /tmp/stegasis/image-%d.jpg -i /tmp/stegasis/audio.
    mp3 -codec copy -c:a aac -strict -2 -b:a 192k -shortest output.mkv
```

Listing 3.15: `FFmpeg` video reassembly command.

The use of `-vcodec copy` tells `FFmpeg` to losslessly extract each `JPEG` image from the `MJPEG` stream - preserving the embedded data.

As with the core project, a video decoder performing the above proposed method was implemented satisfying the `Video Decoder` interface. 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, the above 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 method is to implement embedding algorithms operating on `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 are all based on the `JPEG` DCT LSB method (also known as the *JSteg* algorithm) and 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 are developed including permuted and encrypted variants, reusing ideas and code from the core implementation. Algorithm 3.10 shows pseudocode for the basic embedding algorithm.

Algorithm 3.10 Basic `JPEG` embedding algorithm.

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

Listing 3.17 shows the permuted variant of the implemented `JPEG` embedding algorithm.

The component to embed within is chosen using the calculation `(co % 2) + 1`, this will uniformly distribute 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 **??** illustrates this when a capacity setting of 100% was used.

However, at lower capacity settings it is not possible to visually differentiate between the cover and stego objects[28]. Figure **??** shows the same frame as above, but using a capacity setting of 20%.

[28]This claim is verified within the evaluation section.

```
virtual void embed(Chunk *c, char *data, int dataBytes, int offset) {
  int frameByte = lcg.map[offset++];
  int row, block, co, comp;
  JBLOCKARRAY frame;
  for (int i = 0; i < dataBytes; i ++) {
    for (int j = 7; j >= 0; j --) {
      this->getCoef(frameByte, &row, &block, &co);
      comp = (co % 2) + 1;
      frame = (JBLOCKARRAY)c->getFrameData(row, comp);
      if ((((1 << j) & data[i]) >> j) == 1) {
        frame[0][block][co] |= 1;
      } else {
        frame[0][block][co] &= ~1;
      }
      frameByte = lcg.map[offset++];
    }
  }
};
```

Listing 3.17: Permuted `JPEG` embedding algorithm (`steg/dctp_algorithm.cc:48`).

The `JEPG` embedding algorithms allow `Stegasis` to achieve the >100% embedding capacities claimed within the opening paragraphs of this document. The reason this is possible is due to the final stage of `JPEG` compression - "Encode the resulting coefficients using a Huffman variable word-length algorithm". This effectively compresses the embedded data before it is written back to disk. To show this in action, listing 3.18 shows a 200 MB file embedded within a 100 MB video.

```
$ ls -h
video.mkv 100 MB
$ stegasis mount -alg=dctp -pass=password ./video.mkv /mnt/video
Video mounted at /mnt/video
$ ls /mnt/video
file 200 MB
```

Listing 3.18: Demonstration of 200% embedding capacity.

The above described additions to `Stegasis` allow it to 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.

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 3.19.

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

Listing 3.19: `FUSE` mkdir implementation (`fs/stegfs.cc:158`).

Changes will have to be made to a few `FUSE` functions including `readdir`. 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 3.11 describes the `readdir` implementation.

Algorithm 3.11 Algorithm for the `readdir` implementation.

```
1: add('..')
2: add('.')
3: pathSlashes ← number of slashes in path
4: for for file in (fileSizes and dirs) do
5:     fileSlashes ← number of slashes in file
6:     if path == / then
7:         if file contains one slash then
8:             add(file)
9:     else if path is a prefix of file and pathSlashes == fileSlashes - 1 then
10:        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 of the same video. It was chosen 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. 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

Haven't done this yet...

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 only 0.1% 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.

It was therefore deemed not worth perusing this extension task.

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 11 embedding algorithms and associated code. Listing 3.20 gives an example unit test.

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

Listing 3.20: Embedding algorithm unit test (`steg/steg_algorithm_test.cc:100`).

It was deemed 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 using 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 ?? for some more testing code samples.

4 ‖ Evaluation

4.1 Satisfaction of Requirements

Whoo did everything. also website thing.

4.2 Correctness

Pretty correct.

4.3 Security

Not that secure but can't see.

4.4 Performance

Pretty graphs of time vs algorithm...

5 ‖ Conclusions

Project went really well.

5.1 Future Project Directions

More secure algorithms, cross platform to windows.

References

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