Scott Williams

Computer Science Part II Project Dissertation

STEGANOGRAPHIC FILE SYSTEMS WITHIN VIDEO FILES

Christ's College
University of Cambridge

December 22, 2014

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. Basic 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. The `AVI` file structure, `JPEG` compression and file systems 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,

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

2.1.2 Steganalysis

Steganalysis is the study detecting messages embedded using steganographic techniques; this is analogous to cryptanalysis applied to cryptography. A steganalysis attack is considered successful (i.e. 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 looks at raw uncompressed `AVI` files. Furthermore, only `AVI` version 1.0[6] files are investigated and therefore supported natively[7] by `Stegasis`. Henceforward, references to "`AVI` files" specifically refer to version 1.0, uncompressed.

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

[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 ???

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

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:

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 ???. 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:

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.

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

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 Filesystem in Userspace

need some b.g. info on FUSE.

2.2 Existing tools

The relatively little work on steganography within video is 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.

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.

[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.

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

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.

2.2.4 StegoVideo

StegoVideo is a Virtual Dub filter[12] which allows a user 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

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

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 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% of the cover object's size[14].

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,

[13] 2,600 bytes within a 60 MB video.

[14] This is very much a trade off - capacities upwards of 100% will come at the sacrifice of steganogrphic security. However, this decision is presented to the user rather than decided by the program itself.

$$\forall\ \mathbf{c}, \mathbf{k}, \mathbf{m}.\ Ext_{\mathcal{A}}(Emb_{\mathcal{A}}(\mathbf{c}, \mathbf{k}, \mathbf{m}), \mathbf{k}) = \mathbf{m}.$$

  - 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.

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 partially 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

C because it is web scale refering to the above req.

3 || Implementation

3.1 Introduction

3.2 Filesystem

3.3 Steganographic Algorithms

3.4 Extensions

4 || Evaluation

4.1 Satisfaction of Requirements

4.2 Correctness

4.3 Security

4.4 Performance

5 || Conclusions

5.1 Future Project Directions

References

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