A Steganographic Block Store Across Several Media Files

Christopher Llanwarne[1]

[1]Corpus Christi College, Cambridge, UK

chris.llanwarne@cantab.net

Abstract. *The aim of this work was to create a system for covert storage and retrieval of data in a directory of media files, as an extension of the ability to hide data covertly within a single media file.*
The owner of the data should be able to retrieve the data knowing only which directory to look in, and the key used. The data should be invisible to a casual observer, and difficult for an attacker to detect.

1. Background and Motivation

This introduction sections gives a brief overview of the aims of the project.

1.1. Project Overview

The focus of the project was to producing a working system to store data so that data should be as difficult as possible to detect, it should be possible to deny that any data is being stored at all, any hidden data should be securely stored, and hidden data should be recoverable in the face of media file loss or corruption.

1.2. Motivation

The motivations for holding large quantities of data undetectably are numerous. Amongst these, the ability to ensure freedom of speech and expression can not be arbitrarily restricted by the government of the day is very important.

In a country in which certain types of information are restricted, or entirely outlawed, with draconian punishments, such a system could be invaluable. In addition, the ability to move restricted material past draconian border crossings may prove a valuable tool for journalists reporting from troubled areas.

1.3. Steganography

At its core, steganography is the process of concealing data ‘in plain sight’. That is, it concerns itself not with encoding information so that an antagonist is unable to read it, but rather with preventing the antagonist from realising the information is there at all, even with full access to the medium in which it is stored.

1.4. Related Work: Watermarking

One well-documented[Berghel and O’Gorman 1996] application of steganography is the inclusion of digital watermarks in media files. Watermarking places little emphasis on being undetectable digitally, but rather acts to make sure that the media quality is as unaffected as possible by its inclusion.

This allows the distributing entities to determine whether a media file is a copy of one of their originals. In addition, some more complex watermarks enable the distributor to work out which original copy of their data has been duplicated, so that action can be taken against whoever leaked it.

1.5. Related Work: StegFS

StegFS[McDonald and Kuhn 1999] is a file system proposed as a means to allow someone to plausibly deny that they have data when asked to decrypt it. In order to access a file or directory, the user types in both the name of the object required, and the password to decrypt it. If either the name or the password are unknown, it is not possible to prove that the object exists at all.

By using media noise as the random data I am overwriting, I aim to avoid much of the suspicion that StegFS might invoke. A disk which has been written with random data is likely to invoke suspicion that some unrevealed data is still hidden. By contrast, even the most paranoid people would have trouble suspecting that everyone who uses media files is secretly hiding information in them.

2. Implementation and Decisions

This section details what was decided before coding began, to ensure that the implementation was as smooth as possible, and what was done to convert the scheme designed above into a working program which can hide data successfully, and provide a block-store style interface for the virtual storage device.

2.1. What an attacker can see

An attacker will only ever see the media files. The attacker will never be able to see any internal system state.

In addition, the attacker will only ever see the media files once. Without this assumption, detecting the steganography would be a trivial matter of checking for binary differences in the two copies of the files which the attacker has.

Obviously this requires that the attacker does not see the original images to compare them.

2.2. What an attacker will know

As with all steganography, it is assumed that an opponent will not know that data is stored at all, but this does not mean that the attacker will not be suspicious that this system is in use.

In analogy to a ciphertext-only attack in cryptography, we assume that the attacker will never know or be able to choose what is stored.

2.3. What an attacker can do

It is assumed that any attack will be of the following form:

1. The user sets up the system in a directory of media files.
2. The user hands this directory to the attacker.
3. The attacker examines or modifies the directory in some way.
4. If satisfied, the attacker returns the directory to the user.

There are few assumptions about what an attacker will do once they have the files in their possession:

1. They might add, remove, modify or rename files in an attempt to disrupt the scheme.
2. They might attempt to determine statistically that the media files definitely contain hidden data.
3. They might attempt to break the encryption.

One assumption made is that the data is unmodified in a reasonable number of files. It would clearly be possible to disrupt the scheme by overwriting all files with random data, but this could potentially disrupt almost any form of steganography, so is assumed not to be a risk here.

2.4. Modularity

When designing the system, I have taken care to ensure a modular structure. This will protect the project from becoming redundant in the future. I have ensured that both the steganography method and encryption cipher used can be altered if necessary.

In effect, this allows the project to be used on any media format, so long as a method for hiding data in a single file exists. It also allows the block cipher to be replaced if security flaws are found in the cipher currently being used.

2.5. Extensibility of Design

Because of the way modules will interact by calling very generic function names (encrypt(), for example), it will be simple to replace an insecure or undesirable module with one which performs an equivalent role.

2.6. Project Structure

The project will be structured as follows:

- The project will take a directory of media files and present, as a block store, the hidden space exposed.
- The project will provide top level functions to the user, in a C header file.
  - The format function will take a media directory and set it up of use as a single logical hidden block store.
  - The verify function will be able to verify a block store.
  - Read and Write functions will exist to provide access to a block store.
- There will be 'n' redundant data chains, each contained within a different set of media files. These sets are determined when the directory is formatted.
- Within each chain a complete copy of the data is stored.
- Each file will contain a portion of the data for the chain it is part of. The encryption should be different (i.e. new initialisation vectors) for each file.

2.7. Steganography

Given the modularity, the project can be demonstrated using a very simple data type, while leaving the applicability of the project open to any suitable data type.

I had at first aimed to demonstrate the project on a widespread and much used media format such as MP3. However, due to time constraints, I decided to focus more on the framework and less on individual components. Therefore, I decided to demonstrate the project on BMP bitmap files, while ensuring that the framework could be used easily with other media formats, with minimal extra code required.

Bitmaps are a nice format to demonstrate with, as they can provide a constant ratio of hidden data to cover-image data, which makes disk-space calculations easy. In addition, because no compression is performed, the data is immediately ready for manipulation.

These properties are very nice from the point of view of the steganography module writer, but make no difference to the rest of the framework.

The data hidden in bitmaps is hidden in the final bit of every byte of image data. Clearly this manipulation will alter the image data itself, with the possibility that the manipulation is noticable.

My experience, following some simple initial tests, was that this is only really noticable on 1-bit and 8-bit images. With images at higher quality than this, final bits become much more random-looking, and changes to the final bit in every byte become undetectable. A more detailed investigation of this is given in the evaluation, including some limited statistical analysis.

2.8. Encryption

Given that files might go missing or be invalidated, it is sensible to treat each file independantly, and encrypt the data in each separately.

However, it is bad to encrypt the hidden data in every media file in the same way. Doing so might cause the same plaintext to be encrypted to the same ciphertext. This could leave statistical traces in the ciphertext which an adversary might be able to spot.

Another consideration is that the access times should be as low as possible. Here, we worry only about making sure that the time it takes to read a block does not depend on the size of the media file.

For this project the DES algorithm[1][(FIPS) 1999] was implemented.

The reason for choosing DES is that it has a very simple structure, requiring only bit shifts, permutations, and selection boxes. This will allow a rapid prototype to be constructed. Because the project is modular, this DES can be replaced by a stronger algorithm with a larger key, such as Rijndael, the new AES, should more protection be required.

In practice, using anything less than a 128-bit key is unlikely to provide a reasonable enough level of security.

Within files, I use counter mode, with a randomly generated nonce (per file) appended with an incrementing counter for the 'counter block'. This will allow instant accesses to any region within the file, and should provide equivalent security to CBC, so long as the random nonces for different files never collide. The nonces are to be stored at the beginning of every file.

[1]As has become standard, the terms data encryption standard (DES) and data encryption algorithm (DEA) will be considered interchangeable for this report.

2.9. Redundancy

Storing several redundant copies of the same data in different files will mean that, short of catastrophic data loss, the original data should still be recoverable.

The major differences between this project and related redundancy-provisioning systems are:

- The data-holders (that is, files) are atomic, in the sense that they cannot be made bigger or smaller.
- The files are not physically connected in any way. You cannot simply look at the 5th hard disk in your array. All that is seen is a directory of media files, with no guarantee that they are all present and unmodified.
- If the files are arranged in a logical way, such as biggest size to smallest size, then an attacker can disrupt the system by simply deleting the n biggest or smallest files. The system should be safe against an attacker who knows that the system might be being used.

The scheme must take these changes into account. However, it should still be possible to design a scheme which is only a minor alteration to an existing scheme.

The redundancy used will be very similar to that used in RAID 0. There will be a number of seperate chains of data, the exact number of which can be specified by the user with the `REDUNDANTDATACHAINS` option in the definitions header file.

Each of these chains will contain a full copy of the data. For example, there might be three chains which can hold 300, 350, and 320 KB. Each of these must hold a full copy of the data, so the block-store as a whole would only be able to hold 300 KB of data.

2.10. Combining the Steganography and Cryptography

This stage produced methods which combine the ability to write to media files, and the ability to encrypt data, to allow encrypted read and write operations from the virtual addresses within the media files.

In effect, this provides an interface to media files as though they were themselves block storage devices. To do this, it reads the correct data from the media file, decrypts the necessary blocks, and returns the data which was requested.

2.11. The Top Level

The top level provides interfaces as C functions for useful operations. The operations supported by the header file are reading and writing of data to an existing block storage in media files, verifying the data across the redundant copies in an existing store, and formatting a directory of media files to create a block store.

Reading and writing takes virtual addresses provided by the user, discovers where the data should be stored using the redundancy module, and then accesses it using the combined steganography and cryptography module.

Hence, the user will see a block storage device, but the data itself is spread over several files, it is properly encrypted so it is non-trivial to read, and it has undergone steganography so that an attacker should not be able to discover that data exists at all.

2.12. Separating Data into Chains

The way that data is seperated into chains when formatting is worthy of explanation. It is important that the chains are fairly well balanced, as the amount of data which can be stored is no more than the full length of the shortest chain.

Files are allocated to chains in the following way:

1. Start with n empty chains and a list of files, sorted by size, biggest first.
2. Assign the largest file in the list to the shortest chain, and remove it from the list.
3. Continue in this way until all files are assigned to a chain, and the list is empty.
4. Each chain is shuffled so that it is not deterministic which files will contain which data addresses.

Although a good heuristic, this is not completely optimal, as discussed in the evaluation (Section 3.4).

3. Evaluation

In this section I briefly talk about the security of the framework as a whole, and go on to examine the particular instantiation I implemented.

3.1. Security of the Encryption Scheme

According to NIST[Dworkin 2001], the counter mode of operation provides security equal to the underlying block cipher unless the same counter block is encrypted twice. To attempt to avoid this, I add a random nonce to the start of each counter block. Unless a nonce is repeated, which is extremely unlikely, each block will be encrypted in a different way. This makes the encryption scheme as strong as the underlying block cipher.

3.2. Security of the Steganography

Assuming the media noise being overwritten is random, and that the encrypted output overwriting it is also random, it is impossible to tell if steganography is being used, short of attempting to break the encryption.

Questions arise about the validity of these assumptions, and so the user must still be careful when selecting the method of steganography to use, the block cipher to use, and the media to hide data in.

3.3. Resistance to Deletions

It is possible for an attacker who knows that n redundant chains are being used to determine which media files are in which redundant chain.

Even if they know the number of chains, however, this is not a vulnerability, as they will not know the ordering of files within each chain. Hence it is not possible to simply remove n files to destroy the chain.

3.4. Discussion of file chain allocation

The current method of file chain allocation is given in Section 2.12. This is slightly suboptimal.

For example five files of size 1,100, 1,000, 300, 200 and 200 would be split into two chains: $\{1100, 200, 200\}$ and $\{1000, 300\}$. A more optimal packing, $\{1100, 300\}$ and $\{1000, 200, 200\}$, would give a disk size of 1,400 and waste no space.

However, my packing system provides a reasonable, and very quick, heuristic approach to this problem, and I see little need to change it for minor gains in disk space usage.

3.5. Performance of the Project

I performed a number of tests on the system to determine how rapidly it would be able to store and retrieve data. As an example of a plausible system, I include timing graphs for a block store covering 1000 media files, and with 3 redundant chains. The results are shown in Figures 1 and 2.

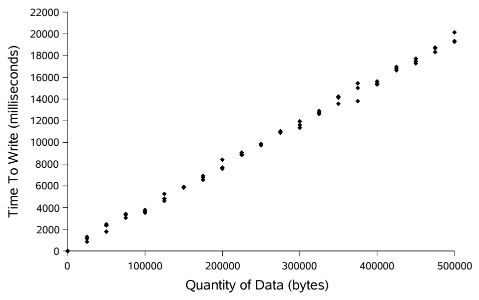

Figure 1. Time Taken to Write Data to a 1000 Media File Block Store with 3 Redundancy Chains

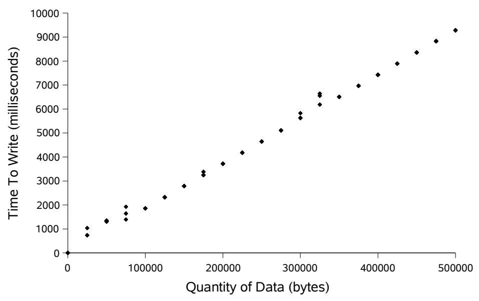

Figure 2. Time Taken to Read Data from a 1000 Media File Block Store with 3 Redundancy Chains

3.6. Visual Quality

In 1-bit images, obviously visible artefacts appear. Because the 8th bit in every byte is being changed, and each pixel represents a byte, this is manifested as visible lines of ‘hidden’ data, particularly noticable in regions of constant colour. An example of this is given in Figures 3 and 4[2].

[2]The corruption does not quite reach the top of the image because I wrote 2000 bytes to a image capable of storing roughly 2150, and bitmap data is stored bottom to top. Artefacts in the left hand image are a result of converting from colour bitmap to monochrome.

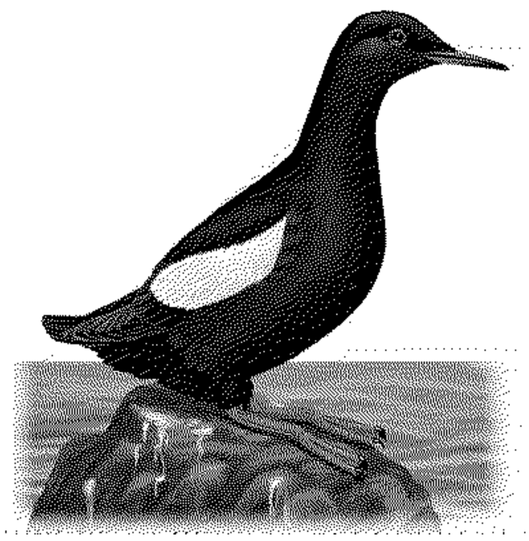

Figure 3. Monochrome Image With No Data Written

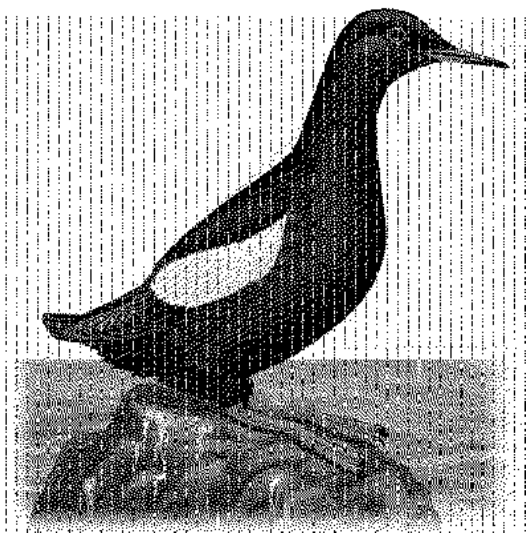

Figure 4. Monochrome Image With Data Written

Moving up to 256-colour images (one byte per pixel), such visual artefacts become not only difficult to see, but almost unnoticable. At 24-bit, monitors and printers are probably unable to display any difference between the two images.

3.7. Statistical Tests

Importantly, statistical tests should be unable to detect that the steganography has occured. This tests the assumption that the underlying noise being overwritten by my steganographic methods looks random. The test which I performed used the linux `ent` program, which performs a series of statistical tests.

On unmodified images, an average entropy of 7.632 bits per byte was recorded as the average, while on images in which every final byte was written with encrypted data, an average entropy of 7.984 bits per byte was recorded

These very similar results suggest that the assumption is sound. However, if photographed data regularly has entropy around 7.6, and encryption has entropy nearer 8.0, statistical tests might cause problems for this system.

4. Further Work and Conclusions

This section briefly touches on areas which would improve the project. Had I had more time, a few of these improvements might have been attempted.

4.1. Dynamic Media File Assignment

It would be nice to allow media files to be assigned every time the project loads, rather than only once, when the directory of media files is initially formatted.

This essentially comes down to reformatting with a new set of media files whenever a new file is added.

4.2. Using a Stream Cipher as a Used-Byte Map

It would be possible to use the output of a stream cipher, using a separate secret key as a map of which potential hidden bits in the media file to actually modify.

For example, if the stream cipher starts with a sequence such as $1011, 0110$, the writing process would only touch data corresponding to the 1st, 3rd, 4th, and so on, hidden bits. The reading process would only ever read from these hidden bits.

4.3. Statistical Analysis of Nearby Hidden Bytes

A potential way to get around the visual artefacts produced in Figure 3 would be to analyse nearby data in the original image to make sure that, for nearby pixels, the values look random. This would mean that the apparenty random data being saved would only ever be written to the media file in regions which look random anyway.

4.4. Nonce Positioning

As it stands, the nonce is always stored at the beginning of every file. This means that by altering a few bytes at the start of every file, the entirety of the data in that file could be lost.

For this project, I have ignored this problem, as I have assumed that the attacker will not modify every image going through. After all, if every image can be modified by an opponent, steganography becomes considerably harder.

One potential solution might be to take a cryptographic hash of the file's header, modulo the number of places the IV could be stored. The IV is then stored at the h^{th} possible location for it in the file, with the data blocks around it.

4.5. Integration with FUSE

FUSE is a linux program which would be able to take my block storage device and integrate it with the linux kernel. This would allow the hidden storage space to be presented to the user by the operating system. This could then be formatted for use with a filesystem such as EXT2, and used as though it were a removable storage device.

4.6. Conclusion

The future work suggested above would provide a greater level of protection against disruption attacks. Although an attacker should not be able to break the encryption, it would not be too difficult to make the scheme unreadable. I imagine that this is a problem to be solved by the steganography rather than the framework developed.

It is my belief that the project designed will allow covert storage of any type of information in media files. In regions in which information is controlled, this could be invaluable, especially given the often draconian punishments for holding controlled information. With the suggested improvements, this could become an extremely useful tool.

References

Berghel, H. and O'Gorman, L. (1996). Protecting ownership rights through digital watermarking. *Computer*, 29(7):101–103.

Dworkin, M. (2001). Recommendation for block cipher modes of operation. `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.

(FIPS), F. I. P. S. (1999). Data encryption standard (des). `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

McDonald, A. D. and Kuhn, M. G. (1999). Stegfs: A steganographic file system for linux. In *Information Hiding*, pages 462–477.