Contents

I Steganography & Steganalysis 1

1 Introduction 3
- 1.1 Introduction to Steganography . . . 3
- 1.2 Introduction to Steganalysis . . . 6
- 1.3 Project Aims . . . 7
- 1.4 About This Paper . . . 7

2 Principles of Steganography and Steganalysis 9
- 2.1 Literature Review . . . 9
- 2.2 Image Processing: JPEG Compression . . . 12
- 2.3 Evaluative Metrics . . . 17

3 Basic Steganography Techniques 19
- 3.1 Steganographic Systems & Definitions . . . 19
- 3.2 Least Significant Bit Substitution Techniques . . . 21
- 3.3 Transform Domain Techniques . . . 27
- 3.4 Conclusion . . . 35

4 Targeted Steganalysis 37
- 4.1 Visual Attacks . . . 37
- 4.2 Structural Attacks . . . 44
- 4.3 Statistical Attacks . . . 47
- 4.4 Conclusion . . . 56

5 Blind Steganalysis 59
- 5.1 Early Methods for Blind Steganalysis . . . 59
- 5.2 JPEG Calibration . . . 60
- 5.3 Blockiness . . . 62
- 5.4 Forming An Attack Through Calibration And Blockiness . . . 64
- 5.5 Conclusion . . . 71

6 Research Conclusion 73

II Software Development 75

7 Introduction 77
7.1 Overview . . . 77
7.2 Disclaimer . . . 77
7.3 Intentions & Considerations . . . 77
7.4 Development Objectives . . . 78
7.5 Development Tools . . . 78

8 API Documentation 79
8.1 Overview . . . 79
8.2 Design Principles . . . 79
8.3 Embedding Functions . . . 80
8.4 Extracting Functions . . . 81
8.5 Attack Functions . . . 82
8.6 Auxiliary Functions . . . 83

9 User Documentation 85
9.1 Overview . . . 85
9.2 Requirements . . . 85
9.3 Interface Arrangement . . . 85
9.4 Using The System . . . 86
9.5 Navigation . . . 92
9.6 Adding New Functions . . . 93
9.7 Bulk Processing . . . 94

10 Testing 95
10.1 Overview . . . 95
10.2 Bit-level Testing . . . 95
10.3 Attack Testing . . . 103
10.4 JPEG Calibration . . . 106
10.5 Summary . . . 106

11 Software Development Conclusion 109

III Project Conclusion 111

12 Project Conclusion 113
12.1 Achievements . . . 113
12.2 Strengths and Weaknesses . . . 114
12.3 Problems Encountered . . . 114
12.4 Reflections . . . 115
12.5 Recommendations For Future Work . . . 115

Part I

Steganography & Steganalysis

1 Introduction

1.1 Introduction to Steganography

1.1.1 What is Steganography?

The term *Steganography* refers to the art of covert communications. By implementing steganography, it is possible for Alice to send a secret message to Bob in such a way that no-one else will know that the message exists. Typically, the message is embedded within another object known as a *cover Work*, by tweaking its properties. The resulting output, known as a *stegogramme* is engineered such that it is a near identical perceptual model of the cover Work, but it will also contain the hidden message. It is this stegogramme that is sent between Alice and Bob. If anybody intercepts the communication, they will obtain the stegogramme, but as it is so similar to the cover, it is a difficult task for them to tell that the stegogramme is anything but innocent. It is therefore the duty of steganography to ensure that the adversary regards the stegogramme - and thus, the communication - as innocuous.

One of the oldest examples of steganography dates back to around 440 BC in Greek History. Herodotus, a Greek historian from the 5th Century BC, revealed some examples of its use in his work entitled *"The Histories of Herodotus"*. One elaborate example suggests that Histaeus, ruler of Miletus, tattooed a secret message on the shaven head of one of his most trusted slaves. After the hair had grown back, the slave was sent to Aristagorus where his hair was shaved and the message that commanded a revolt against the Persians was revealed [19]. In this example, the slave was used as the carrier for the secret message, and anyone who saw the slave as they were sent to Aristagorus would have been completely unaware that they were carrying a message. As a result of this, the message reached the recipient with no suspicion of covert communication ever being raised.

In modern terms, steganography is usually implemented computationally, where cover Works such as text files, images, audio files, and video files are tweaked in such a way that a secret message can be embedded within them.

The techniques are very similar to that of *digital watermarking*, however one big distinction must be highlighted between the two. In digital watermarking, the focus is on ensuring that nobody can remove or alter the content of the watermarked data, even though it might be plainly obvious that it exists. Steganography on the other hand, focuses on making it extremely difficult to tell that a secret message exists at all. If an unauthorised third party is able to say with high confidence that a file contains a secret message, then steganography has failed.

Steganography also differs from *cryptography* because the latter does not attempt to

hide the fact that a message exists. Instead, cryptography merely obscures the integrity of the information so that it does not make sense to anyone but the creator and the recipient. The adversary will be able to see that a message exists, and the inverse process of *cryptanalysis* involves trying to turn the meaningless information into its original form.

With this said, it is still highly likely that a complete steganographic system might employ cryptographic measures as a safety-net to protect the content of the message in the event that the steganography is broken.

1.1.2 How is Steganography Used?

When a steganographic system is developed, it is important to consider what the most appropriate cover Work should be, and also how the stegogramme is to reach its recipient. With the Internet offering so much functionality, there are many different ways to send messages to people without anyone knowing they exist. For example, it is possible that an image stegogramme could be sent to a recipient via email. Alternatively it might be posted on a web forum for all to see, and the recipient could log onto the forum and download the image to read the message. Of course, although everyone can see the stegogramme, they will have no reason to expect that it is anything more than just an image.

In terms of development, Steganography is comprised of two algorithms, one for embedding and one for extracting. The embedding process is concerned with hiding a secret message within a cover Work, and is the most carefully constructed process of the two. A great deal of attention is paid to ensuring that the secret message goes unnoticed if a third party were to intercept the cover Work. The extracting process is traditionally a much simpler process as it is simply an inverse of the embedding process, where the secret message is revealed at the end.

The entire process of steganography for images can be presented graphically as:

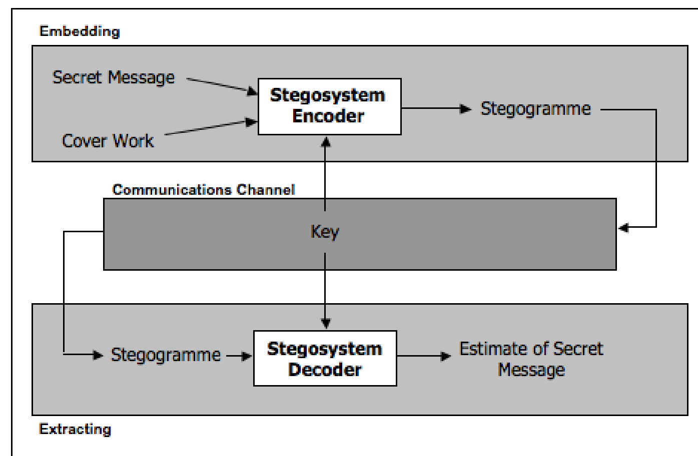


Figure 1.1: The Process of Steganography.

Figure 1.1 shows one example of how steganography might be used in practice. Two

inputs are required for the embedding process:

1. **Secret message** - usually a text file that contains the message you want to transfer

2. **Cover Work** - used to construct a stegogramme that contains a secret message

The next step is to pass the inputs through the *Stego-system Encoder*, which will be carefully engineered to embed the message within an exact copy of the cover Work, such that minimum distortion is made; the lower the distortion, the better the chances of undetectability. The stego-system encoder will usually require a *key* to operate, and this key would also be used at the extraction phase. This is a security measure designed to protect the secret message. Without a key, it would be possible for someone to correctly extract the message if they managed to get hold of the embedding or extracting algorithms. However, by using a key, it is possible to randomise the way the stegosystem encoder operates, and the same key will need to be used when extracting the message so that the stegosystem decoder knows which process to use. This means that if the algorithm falls into enemy hands, it is extremely unlikely that they will be able to extract the message successfully.

The resulting output from the stego-system encoder is the *stegogramme*, which is designed to be as close to the cover Work as possible, except it will contain the secret message. This stegogramme is then sent over some communications channel along with the key that was used to embed the message. Both the stegogramme and the key are then fed into the *stego-system decoder* where an estimate of the secret message is extracted. Note that we can only ever refer to the output of the extraction process as an *estimate* because when the stegogramme is sent over a communications channel, it may be subjected to noise that will change some of the values. Therefore, we can never be sure that the message extracted is an exact representation of the original. Also, the recipient will obviously never know what the original message was, and so they have nothing to compare it to when it is extracted.

This is probably the most common system of image steganography today, with the focus mainly on developing the stego-system encoder carefully. It is of paramount importance in steganography that the stegogramme contains no trail of embedding a secret message if it is to be successful.

In recent years, many steganographic algorithms have been made publicly available, and so it is very easy for anyone with even a limited knowledge of steganography to be able to communicate covertly. Most of the systems make use of everyday images as the basis for the models, and of course, the information that is hidden within those images can range from anything between harmless gibberish and messages that are a threat to national security. Subsequently, there is a growing concern as to how we can identify whether any image contains steganography, such that we can be sure the technology is not used for the wrong purposes. This counter-activity is referred to as *Steganalysis*, and much resource and research has been put into determining whether an image is innocent or not.

1.2 Introduction to Steganalysis

1.2.1 What is Steganalysis?

Steganalysis is the art of identifying stegogrammes that contain a secret message. Steganalysis does not however consider the successful extraction of the message, this is usually a requirement for cryptanalysis.

Typically, steganalysis begins by identifying any artifacts that exist in the suspect file as a result of embedding a message. None of the steganographic systems that are known today achieve perfect security [1], and this means that they all leave hints of embedding in the stegogramme. This gives the *steganalyst* a useful way in to identifying whether a secret message exists or not.

1.2.2 How is Steganalysis Used?

Steganalysis is very important to international security, as growing interest emerges as to whether terrorist organisations use steganographic techniques to communicate with each other. In fact steganalysis is taken so seriously that it is believed that US Government agencies, including the NSA and the Pentagon are funding research for its development. If a file is considered to contain a secret message then it is possible that the entire Work will be modified by the steganalyst such that the integrity of the message is removed. This means that that Bob will not be able to make sense of the message that Alice sent to him when he attempts to extract it. A much safer solution however, is that the Work is deleted so that it never reaches the recipient.

Steganalysis is an extremely difficult science, as it relies on insecure steganography. As discussed in section 1.1.1, if steganography is to be successful, it should leave no indication that a secret message exists. Thus, if the model has been created successfully, it should be a difficult task for any third party to spot that tampering has occurred.

Jessica Fridrich [5] suggests that *"the ability to detect secret messages in images is related to the message length"*. This statement is based on the logic that a small message embedded within a large carrier will result in a small percentage of manipulations, and therefore it will be much harder to spot any artefacts within the stegogramme.

Of course, the success of steganalysis also depends on what information the steganalyst has to work from. There are two main classifications of steganography - *targeted*, and *blind*. Targeted steganalysis works when a method designed for identifying a specific steganographic algorithm has been developed [1]. For example, embedding within pixel values leaves patterns that can be searched for with suspicious files. If the steganalyst is sure that covert communications are taking place, and also knows of a possible method for how a secret message can be embedded, then it should be a fairly trivial task to summarise if the file contains this type of steganography or not. Blind steganalysis on the other hand is a much harder task, and means that the steganalyst has no reason to believe that covert communications is taking place. In this case, a set of algorithms are typically developed in order to check for signs of tampering. If some signs of tampering are flagged by the algorithms, then it is likely that the suspect file contains steganography.

1.3 Project Aims

There are two main aims of this project. The first aim is to research the fields of steganography and steganalysis, such that we can present a variety of techniques from both. We will start by discussing several techniques for hiding a secret message within an image, and then consider how this can be detected from a steganalytical viewpoint. The second aim of the project is to develop these research findings into a complete system that is capable of acting as an educational tool for both steganography and steganalysis.

1.4 About This Paper

As this project can be split into two distinct parts, the following report has also been split such that a clear distinction can be made between the research work and the development work. Part I of the report focuses on the research material, and will explain the logic and methodologies behind a wide range of steganographic and steganalytical algorithms from the viewpoint of real life practise. Part II focuses on the development aspect of the project by describing what has been developed, and how the system can be used.

Each part will end with a conclusion on what has been learnt when carrying out each task. This will be followed by an evaluative conclusion at the end of the report which will summarise what has been learnt during the project as a whole.

2 Principles of Steganography and Steganalysis

2.1 Literature Review

In order to understand the methods for steganography and steganalysis that will be discussed later on in the project, it is first necessary to build up a clear picture of both fields.

This chapter will discuss the main principles of steganography and steganalysis by firstly discussing where we currently stand in both fields, and then introducing the necessary background knowledge that is required to properly understand the methods introduced in this project. The JPEG compression process is also discussed as it is essential that we understand the significance of embedding the message data within this domain at a later stage. Finally, we introduce some evaluative metrics so that we can easily relate to the success and failure of the steganalysis techniques in future chapters.

2.1.1 Past Work

Given time, it has been possible to break every steganographic system that has ever been published. As a result of this, new techniques are developed to improve upon the flaws of the predecessor. It is very much the same case in steganalysis where the algorithms are often tweaked or combined in order to attack the latest steganography algorithm. The two fields therefore operate in a 'cat and mouse' style strategy with steganography aiming to be ahead of the field such that covert communications may exist. There is then a call for steganalysis to catch up such that covert communications are minimised as much as possible.

One of the most basic steganographic techniques is achieved by manipulating the pixel values of the cover image in sequence such that they then turn into code that can be used to reconstruct the message when extracting. The most popular method for developing this concept left a pattern in the beginning of the images that steganalysts bought to light. This led to the development of a series of techniques that could detect the existence of steganography within any stegogramme that followed this embedding strategy, and so steganographers set about developing a new system that would render the attacks useless.

This led to the development of randomised embedding in pixel values, and once again steganlalysts found a method for detecting steganography for all images created using this strategy. The chase has continued for hundreds of known steganographic methods, and we are now at the point where a steganography algorithm has been developed that is immune to all the known blind steganalysis methods, it is called YASS (Yet Another

Steganographic Scheme). YASS embeds data into seemingly random locations of a cover image in such a way that no current method is able to spot any artefacts that enable a way in to cracking the algorithm [22].

2.1.2 Steganography Nomenclature

There are many steganographic algorithms that have been developed to allow covert communication, and each of them will be designed to make steganalysis a difficult task by finding where the redundancies are within a cover Work, and then modifying the redundant data such that it holds a secret message.

Steganography can be split into two main categories:

1. *Statistics-aware* steganography
2. *Model-based* steganography

Each of these categories aims to preserve the qualities of the original cover Work, with the logic that it will be hard to detect steganography in suspect Work that is identical to an innocent Work.

Statistics-aware steganography considers the statistical techniques that steganalysts are known to use to detect steganography. With these techniques in mind, the steganographic system is developed in such a way that none of these attacks will prove successful. In other words, the stegogramme that is produced after embedding a secret message will be statistically sound. Ingemar Cox [1] illustrates this by example when he suggests that an image can be "completely described" by its histogram (a graphical chart of pixel frequencies). If this is the case, histograms can be used to evaluate whether any unusual trends take place. A steganographic system that preserves the same histogram as the cover Work would therefore be a prime example of a statistics-aware steganographic implementation.

Model-based steganography considers the preservation of a chosen model of the cover Works, rather than its statistics [1]. This framework measures the components of a cover Work that do not change after embedding, against the components that change as a result of embedding. The resulting task is to develop a steganographic system that ensures the values that change are as minimal as possible.

It is also possible that a steganographic system may be developed such that the values that change as a result of embedding look like natural processing. This would be implemented in the hope that the steganalyst puts the modifications down to processing rather than deliberate manipulation. In practice however, it is a fairly trivial task for a steganalyst to determine the likelihood that the modifications are innocent, and therefore they can easily calculate whether steganography exists or not.

In order to understand more about how a steganographic algorithm is created, it is necessary to first understand exactly what makes a good steganographic image:

Perceptability

The stegogramme that is produced after embedding a secret message, should not be altered such that it is visually obvious that information has been embedded. In fact, the resulting image should be so similar to the original that if you compare both side by side, you should not be able to see any difference at all between the two.

Capacity

The amount of information that is embedded should be as small as possible. Logic suggests that the longer the message, the more the image has to be altered to compensate for this. Obviously, the more a Work is modified, the easier it is for the steganalyst to discover artefacts within an image. Therefore, the usual practise for embedding is to make the message as short as possible so that the image is altered as little as possible.

Robustness

This refers to the degree of difficulty required by a steganalyst to determine whether or not the image contains a hidden message or not. A good implementation of steganography would be one where the image can be subjected to many attacks that each prove inconclusive.

Christy A. Stanley [23] suggests that another important property of steganography is *speed*, where information should be embedded as quickly as possible. However, it is generally considered that this is not as important as those properties that have been outlined. Speed is really more of a constraint than a key principle of steganography. It does not seem feasible that anyone embedding information would sacrifice any of the above-mentioned properties because they need the algorithm to embed information within a timely manner - the consequences are too damning.

If a steganographic algorithm can fulfil these three principles, then the resulting image will highly likely reach its recipient undetected, meaning a successful implementation of steganography has been developed.

2.1.3 Steganalysis Nomenclature

The desired outcomes for steganalysis depend on what the steganalyst wants to achieve. For example, one steganalyst may just want to know whether Alice and Bob are communicating, where as another steganalyst may want to know how Alice and Bob are communicating so that they can impersonate Alice and send Bob false messages. In fact, the role of the steganalyst can be defined in three categories - *passive*, *active*, and *malicious*.

Passive steganalysts intercept a Work as it is passed through the communications channel, and then tests it to identify whether it contains a secret message or not.

If no secret message is detected, the Work will be allowed to continue through the communications channel. However, if a secret message is detected, the steganalyst will block the transmission and Bob will not receive the secret message. However, the fact

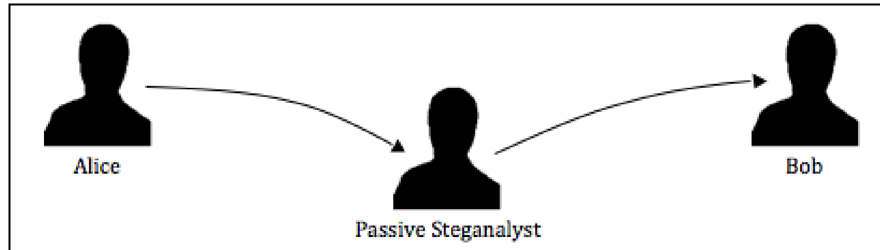


Figure 2.1: The Passive Steganalyst.

that Bob does not receive the message he was waiting for may lead him to suspect that a steganalyst has successfully broken the communication, and therefore they will often change the algorithm and resend the message again.

An active steganalyst differs from a passive steganalyst because if the existence of a secret message is found, the active warden would modify the Work such that the integrity of the message is broken. This modification may be achieved by compressing the image in the hope that some important pixel values alter the secret message data. Most steganographic techniques assume a passive steganalyst, and therefore the stegogramme is not designed to survive modifications such as these. With this method, Bob will still receive the Work, but when he extracts the message he will find it does not make sense.

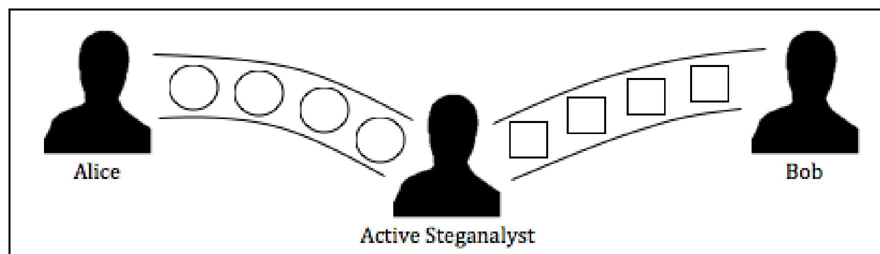


Figure 2.2: The Active Steganalyst.

A mailicious steganalyst will analyse the Work sent between Alice and Bob to determine if it contains a hidden message. If it does, they will then try and work out how the message was embedded such that they can then impersonate Alice and send their own messages to Bob.

However, this method is about much more than just detecting whether or not a secret message is embedded within a Work. For this reason, it is very rare for a steganalyst to carry these traits.

2.2 Image Processing: JPEG Compression

JPEG compression is a commonly used method for reducing the file size of an image, without reducing the aesthetic qualities enough to become noticeable by the naked eye.

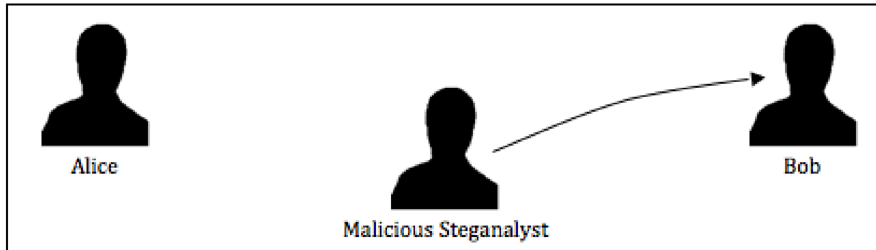


Figure 2.3: The Malicious Steganalyst.

Broadly speaking, it extracts all the information from an image that the human eye is not perceptible to - and would therefore not miss - should it not be there. The compression of JPEG images contains several processes:

1. Converting pixel values to YCbCr
2. Downsampling the chrominance values
3. Transforming values to frequencies
4. Quantisation
5. Zig-Zag ordering
6. Lossless Compression

This section outlines the theory and logic behind each of these steps.

2.2.1 Converting pixel values to YCbCr

The first step is to convert the RGB colour layers of the image into three different components (Y, Cb, and Cr). The Y component relates to the luminance (brightness), and the U and V elements relate to chrominance (colour). The chrominance coefficients of an image (Cb and Cr) are determined by a 2D grid that has blue to yellow on one axis, and red to green on another.

2.2.2 Downsampling the chrominance values

The second phase of JPEG compression is to downsample[1] (sometimes referred to as subsampling) the image. It is a common belief that the human eye is more sensitive to changes in brightness than to changes in colour [13]. This means that it is possible to remove a lot of colour information from an image without losing a great deal of quality. As a result of this, much of the compression takes place by downsampling the chrominance data to reduce the overall file size. Incidentally, audio compression takes place in much the same way where a lossless WAV file is compressed to lossy MP3 formats by downsampling the frequencies that are out of the range of the human ear.

[1]Downsampling is the process of reducing the quality of a signal with the goal of making the file size smaller [Source: http://en.wikipedia.org/wiki/Downsample]

JPEG compression often downsamples by taking 4 adjacent pixels and averaging them to one value. Doing this for the entire image will remove $(\left[\frac{1}{3} \times \frac{3}{4}\right] \times 2)$ of the image information which is roughly half [19]. All of this can be done without any noticeable drop in the quality of the image.

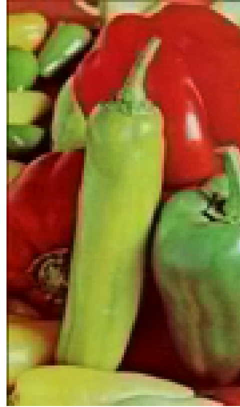
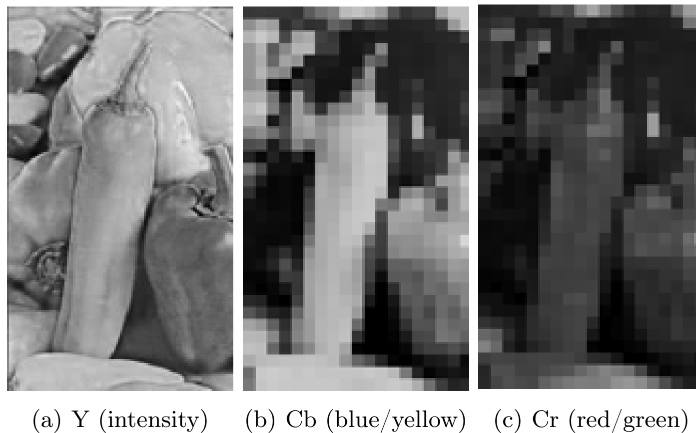


Figure 2.4: An image compressed using the JPEG compression algorithm [17].

Figure 2.4 shows an image that has been compressed using JPEG with a factor of 8. This means that the colour information for this image is comprised of 8x8 blocks of pixels. The quality of this file seems perfectly acceptable to the human eye, but let us take a look at how the image got to this state.



(a) Y (intensity) (b) Cb (blue/yellow) (c) Cr (red/green)

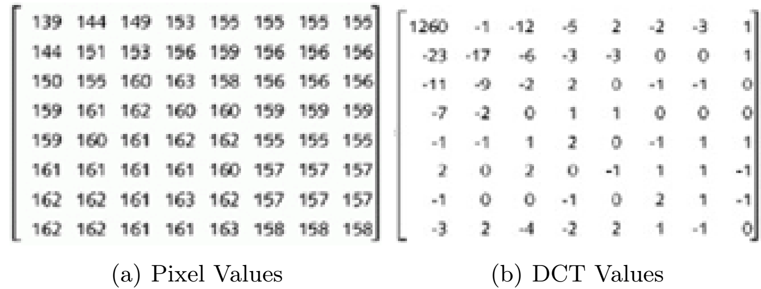
Figure 2.5: The three channels for JPEG Compression

Figure 2.5 shows each channel of the image. Hopefully what is clear is that the Cb and Cr values can be dramatically reduced without any noticeable loss of quality to the image.

2.2.3 Transforming values to frequencies

The Discrete Cosine Transform (DCT) is used for JPEG images to transform them into frequencies. DCT is a mathematical transform (typically a cosine function) that converts the pixels by seemingly 'spreading' the location of the pixel values over part of the image. It does this by grouping the pixels into 8 x 8 blocks and transforming them from 64 values into 64 frequencies (DCT coefficients. By modifying just a single DCT coefficient, the entire 64 pixels in that block will be affected.

To illustrate how this process changes the results, consider Figure 2.6.



$$\begin{bmatrix} 139 & 144 & 149 & 153 & 155 & 155 & 155 & 155 \\ 144 & 151 & 153 & 156 & 159 & 156 & 156 & 156 \\ 150 & 155 & 160 & 163 & 158 & 156 & 156 & 156 \\ 159 & 161 & 162 & 160 & 160 & 159 & 159 & 159 \\ 159 & 160 & 161 & 162 & 162 & 155 & 155 & 155 \\ 161 & 161 & 161 & 161 & 160 & 157 & 157 & 157 \\ 162 & 162 & 161 & 163 & 162 & 157 & 157 & 157 \\ 162 & 162 & 161 & 161 & 163 & 158 & 158 & 158 \end{bmatrix}$$

(a) Pixel Values

$$\begin{bmatrix} 1260 & -1 & -12 & -5 & 2 & -2 & -3 & 1 \\ -23 & -17 & -6 & -3 & -3 & 0 & 0 & 1 \\ -11 & -9 & -2 & 2 & 0 & -1 & -1 & 0 \\ -7 & -2 & 0 & 1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 2 & 0 & -1 & 1 & 1 \\ 2 & 0 & 2 & 0 & -1 & 1 & 1 & -1 \\ -1 & 0 & 0 & -1 & 0 & 2 & 1 & -1 \\ -3 & 2 & -4 & -2 & 2 & 1 & -1 & 0 \end{bmatrix}$$

(b) DCT Values

Figure 2.6: Pixel Values vs. DCT coefficients

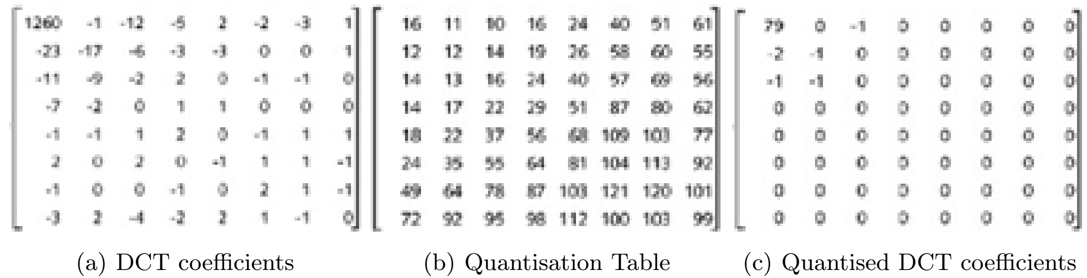
Figure 2.6 shows an example of the effects of applying DCT to an image. The left part of the diagram is an 8x8 block of image data. It could be either luminance or chrominance data - at this point it does not matter. The image on the right is the result after applying DCT to this block of the image. Note how the largest value is located in the top-left corner of the block, this is the actually the lowest frequency. The value is so high because the data is encoded with the highest importance and the lowest frequency [17]. This, in simpler terms means that this value is the average value of all the pixel values in this block. The values are typically always high around the top-left corner of a DCT block but note that the numbers closest to zero seem to populate around the lower-right corner of the block. These are the high frequencies and it is these values that are removed during the next step.

If we were to precisely inverse the DCT function that was applied to yield the results on the right of Figure 2.6, we would end up with the original results (left image of Figure 2.6).

2.2.4 Quantisation

The next step is arguably the most important one when compressing JPEG images. The aim is to quantise the values that represent the image after stage 3. Quantisation is the process of taking the remaining coefficients and dividing them individually against a pre-determined set of values and then rounding the results to the nearest real number value. The higher these pre-determined values are determines how much detail will be removed from the image. The higher the numbers, the more detail is eliminated. The goal is

to eliminate the high frequency (lower-right) values. Imagine you have a birds-eye view on a dense collection of trees. There may be smaller trees that are situated underneath the trees that you can see, but as you cannot see them, it would not affect your view whether they were there or not. This is exactly the same principle for quantisation - by eliminating the values that would not make a difference to the image should they not exist, a great deal of data can be removed with minimum effect on the quality.



1260	-1	-12	-5	2	-2	-3	1
-23	-17	-6	-3	-3	0	0	1
-11	-9	-2	2	0	-1	-1	0
-7	-2	0	1	1	0	0	0
-1	-1	1	2	0	-1	1	1
2	0	2	0	-1	1	1	-1
-1	0	0	-1	0	2	1	-1
-3	2	-4	-2	2	1	-1	0

(a) DCT coefficients

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(b) Quantisation Table

79	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(c) Quantised DCT coefficients

Figure 2.7: The Quantisation process.

Image (c) shows the result of quantising the original DCT coefficients (a) with the quantisation table (b). It is within these quantised DCT coefficients that transform steganography operates, as we will see in Chapter 3.

2.2.5 Zig-Zag ordering

The quantised DCT coefficients shown in 2.7 are fairly typical. There are only a few values that hold numbers other than zero - the majority will always be zeros. It is also typical to see the non-zero numbers in the upper left, and zeros as you get towards the lower-right corner. As this is the case, this stage of JPEG compression reorders the values using a 'zig-zag' type motion so that similar frequencies are grouped together.

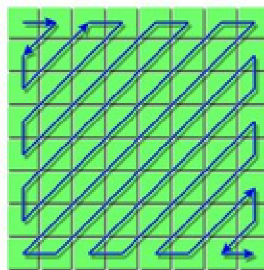


Figure 2.8: *zig-zag* ordering of JPEG image components

2.2.6 Lossless Compression

The last process involves the use of two different algorithms. It is beyond the scope of this report to go into detail about how the algorithms work, but broadly speaking the first algorithm, 'Run-Length Encoding' (RLE) compresses the high frequency coefficients and a 'Differential Pulse Code Modulation' (DPCM) compresses the first low frequency coefficient. A Huffman algorithm is then used to compress everything. Finally, the Huffman trees are stored in the JPEG header.

Now that we have looked into how JPEG compression works, it should be easier to understand a bit more about how the transform-based stego-systems operate in Chapter 3.

2.3 Evaluative Metrics

When evaluating the success or failure of steganalysis algorithms, they are often classified based on how many stegogrammes are successfully identified, and how many pass through undetected. The terms *false-negatives* and *false-positives* are used to express this.

False-negatives refer to the amount of stegogrammes that are not detected when applying the steganalysis algorithm. In other words, it expresses an indication of how reliable the algorithm is. If there are many false-positives, then the algorithm should not be depended on for filtering steganographic content.

The term *false-positive* means the exact opposite, where the algorithm deems innocent Works to be stegogrammes.

A good steganalytical model minimises both error probabilites. Thus, by measuring each of these variables, we can see how successful the algorithm is. However, the probability of error is likely to depend on a number of factors including the type of image that has been used as a cover. It may therefore also be necessary to measure error rates for different embedding properties in order to successfully determine how successful the algorithm is as a whole.

3 Basic Steganography Techniques

In order to understand more about how steganography works - and indeed to raise awareness of the magnitude of known embedding techniques - we will discuss some of the most documented steganographic approaches, splitting them into appropriate groups to illustrate clearly how they differ from each other.

The following techniques are just a handful of the earliest examples of image steganography. Some are easier to implement than others, and some are more robust than others. This - as we will see later on in this phase of the project - is one of the biggest trade-off's with steganography. A simple algorithm is likely to be just as simple to break, whilst the more clever algorithms are harder to break. This is an example of *security by obscurity*; something that is fairly rare to see in computer security. The question steganographers usually face is: "*how well do you want to hide the existence of a secret message?*". If the message just needs to be hidden from one person with little computing experience, then a simple algorithm may be preferred. However, terrorist cells would be looking to hide the existence of the message from government defence agencies typically comprised of some of the most highly skilled people in computing. In which case a clever algorithm is likely to be preferred, certainly something beyond the scope of this entire project.

3.1 Steganographic Systems & Definitions

Before diving deep into the core of each algorithm, we should take a moment to define the elements of both the encoding and decoding stages of steganography (common across all embedding algorithms). In order to explain the processes easily later on in the chapter, we will assign mathematical representations to the elements. By doing this, we should hopefully improve the readability, and make it easier to understand how each algorithm works with regards to others.

3.1.1 The Stego-system

A *stego-system* (steganographic system) in image steganography refers to a system capable of hiding a secret message within an image, such that no third-parties are aware that the message exists. The image that is output from this process is known as a *stegogramme*, and great care is taken to ensure that this looks as innocent as possible so that the secret message has the best chance of reaching its intended recipient.

However, the stego-system not only refers to encoding the message, it also refers to the system that makes it possible to read the message when it reaches its recipient. The two sub-systems are referred to as the *stego-system encoder* and the stego-system decoder respectively.

3.1.2 The Encoder

The *stego-system encoder* is the heart of the steganographic system. It makes it possible to embed a secret message within some cover medium. In the case of image steganography, the encoder will read in a cover image $c_{m,n}$ (where m and n refer to the height and width dimensions of c), and will embed a message m by tweaking carefully selected values of the cover image c_i. Exactly which c_i values are tweaked depends on the specific embedding algorithm, and it is this decision process that makes each stego-system unique.

The regions that are typically chosen to hide the message, are often selected because they are believed to hold *redundant data*. That is to say that replacing the image data with the message data has no direct impact on the overall perceptibility of the image, therefore meaning the message is hard to detect - at least with the naked eye.

Figure 3.1 shows a graphical representation of the elements and processes typically associated with a stego-system encoder.

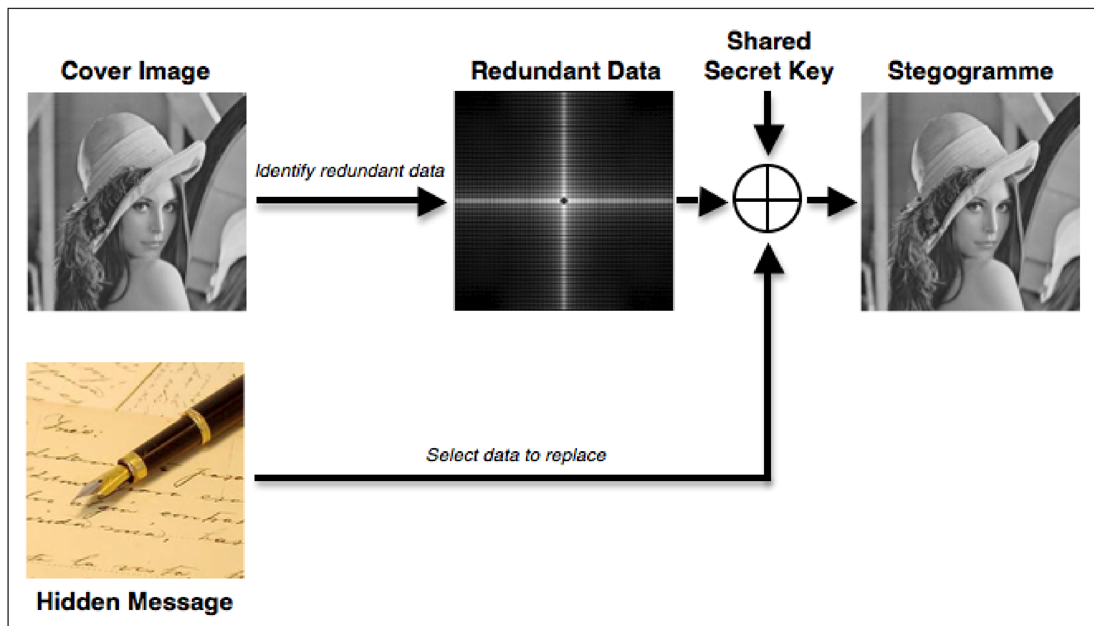


Figure 3.1: The general design of a basic stego-system encoder (adapted from [19]).

As the illustration suggests, cover image c is read into most stego-systems such that the redundant data can be identified. By using a shared key k, the encoder references the exact locations of the regions that have been replaced with the data from m. This makes it possible to select the correct regions of the stegogramme and retrieve the secret message successfully when decoding (see section 3.1.3). Stego-systems such as these are referred to as *secret key stego-systems*.

Of course, how many values are tweaked in c to produce the stegogramme depends on the length of the message $l(m)$. In order to function correctly, $l(m)$ must be less than or

equal to the amount of redundant data available for the image.

From time to time in this chapter we will need to refer to each value of m as individual units. This will be denoted by m_i, where each element can be assumed to be binary values, $m_i \in \{0, 1\}$.

3.1.3 The Decoder

The *stego-system decoder* allows the receiver of the stegogramme to obtain an *estimate* of the secret message m'. We refer to the output message as an estimate because each m_i is derived from the locations of the stegogramme according to the shared key k. This means that it is not the exact same message that was input into the encoder, so we can never say that it is 100% identical; indeed, some of the values may be slightly off. However, a good stego-system ensures that the estimate will be as close to the original message m as possible.

Figure 3.2 shows a graphical representation of the elements and processes associated with a common stego-system decoder.

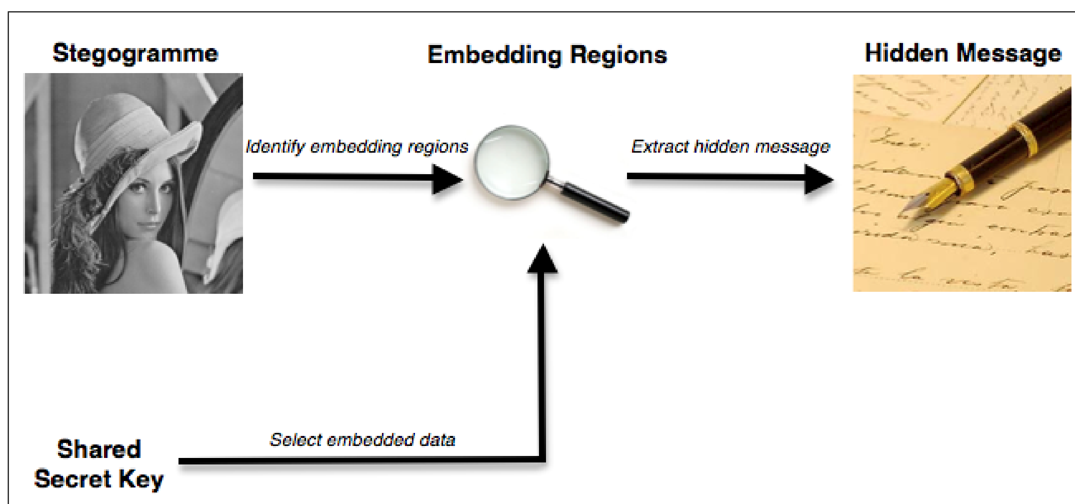


Figure 3.2: The general design of a basic stego-system decoder.

As we can see, both the stegogramme and shared secret key k are used to identify the regions that hold the message data. When both of these elements are provided, each m_i can be retrieved such that the complete binary sequence m' can be constructed. The message data can then be converted to an alternative form (typically ASCII) so that it can be read.

3.2 Least Significant Bit Substitution Techniques

One of the earliest stego-systems to surface were those referred to as *Least Significant Bit Substitution* techniques, so called because of how the message data m is embedded

within a cover image c. In computer science, the term *Least Significant Bit (LSB)* refers to the smallest (right-most) bit of a binary sequence. The structure of binary is such that each integer may only be either a 0 or a 1, often thought of as *off* and *on* respectively. Starting from the right, the value (if on) denotes a 1. The value to its left (if on) denotes a 2, and so on where the values double each time. Now let us consider the following 8-bit binary sequence:

1 0 1 1 0 0 1 **1**

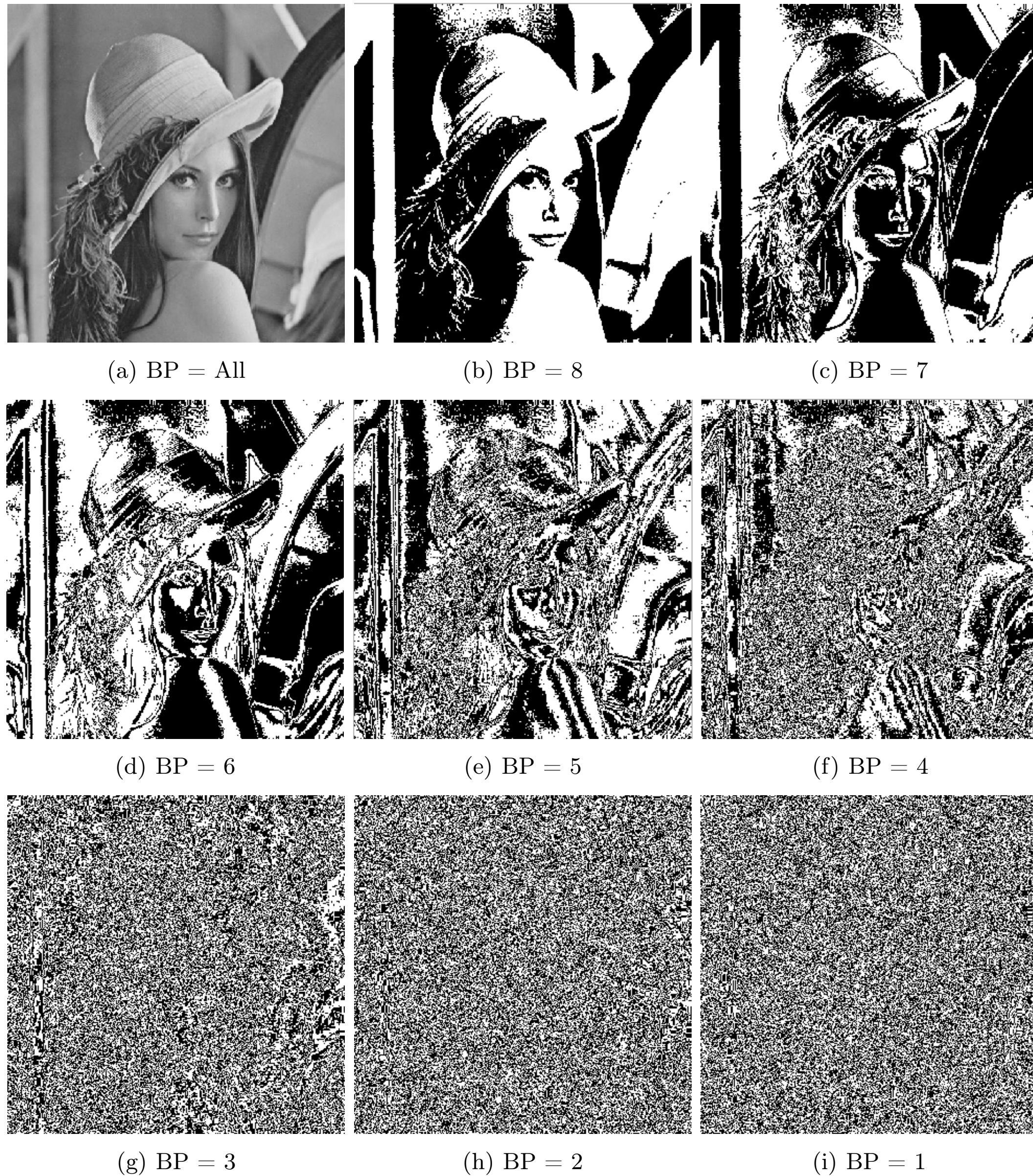
Summing all the values equal to 1 yields a result of 179. The right-most value (denoted in **bold** text) is the LSB of this sequence. This value essentially determines whether the total sum is odd or even. If the LSB is a 1, then the total will be an odd number, and if 0, it will be an even number. However, changing the LSB value from a 0 to a 1 does not have a huge impact on the final figure; it will only ever change by +1 at most.

If we now think of each 8-bit binary sequence as a means of expressing the colour of a pixel for an image, it should be clear to see that changing the LSB value from a 0 to a 1 will only change the colour by +1 - a change that is unlikely to be noticed with the naked eye. In fact, the LSBs of each pixel value could potentially be modified, and the changes would still not be visible. This highlights a huge amount of redundancy in the image data, and means that we can effectively substitute the LSBs of the image data, with each bit of the message data until the entire message has been embedded. This is what is meant by Least Significant Bit Substitution.

Finally, when we talk of Least Significant Bit Substitution algorithms, we should mention that this encompasses two different embedding schemes: *sequential* and *randomised*. Sequential embedding often means that the algorithm starts at the first pixel of the cover image $c_{0,0}$ and embeds the bits of the message data in order until there is nothing left to embed. Randomised embedding however, scatters the locations of the values that will be modified to contain the bits of the message data. The main reason for randomising the approach is to make things a little trickier for the steganalysts that are looking to determine whether the image is a stegogramme or not. The implications of embedding in this fashion over the sequential approach are discussed in detail in Chapter 4.

3.2.1 Overview

In the early stages of image steganography development (as we know it today), many steganographers believed that the least significant bits of an image were an ideal place to embed the message data, not only because their modification yields no perceptible loss of quality, but also because they believed the LSBs were completely random in terms of their overall significance to the complete image. In other words, it was common belief that if the LSBs of an image were viewed in isolation as a binary image (where 0 = black, and 1 = white) then the distribution will appear so scattered that modifying the values will make no difference to its appearance - it would still look very random. Figure 3.3 illustrates why this assumption was made. The Figure shows a grayscale image (a) and allows for a comparison of each of its bit planes (b) → (i), where "BP = 8" corresponds to the most significant bit plane, and "BP = 1" corresponds to the LSB plane.



(a) BP = All (b) BP = 8 (c) BP = 7

(d) BP = 6 (e) BP = 5 (f) BP = 4

(g) BP = 3 (h) BP = 2 (i) BP = 1

Figure 3.3: An image and each of its bit planes (BP) in descending order.

When we look at each bit plane in this manner, it does appear as though the LSB plane in (i) is more random than that of a bit plane higher up in the scale such as (d), thus it is understandable why such an assumption was made. However, Andreas Westfeld and Andreas Pfitzmann [26] found that this hypothesis was incorrect. Their works suggests that the LSBs - whilst perhaps random in terms of appearance - are no more so than any other bit plane in terms of design. If we consider an image of natural life (thus

disregarding artificial or computer-based sketches), the image is almost guaranteed to include objects that contain gradual colour changes due to natural filters such as light, shadows, and the texture of the object itself. Typically, a shadow that is cast on an object for example, produces a pattern in the pixel values such that the values in that area of the image decrease by very small amounts as the shadow gets stronger. It is also suggested by Wayner that some camera's pad the data by adding extra detail to produce 24-bit images [24], and it is also true that JPEG compression incorporates averaging that may result in large areas of the image having the same LSBs. It therefore seems to be the case that the LSBs of an image are more structured than the steganographers originally believed, and this poses a huge weakness in the authors' systems that can be exploited through visual attacks. We will return to this point later on in section 4.1, but for now let us just accept that the LSBs were chosen as an ideal region for embedding the message data.

3.2.2 Hide & Seek: The Sequential Approach

The simplest form of image steganography is the method known as *Hide & Seek* which replaces the LSBs of pixel values (also referred to as the *spatial domain*) with the bits from the message bit stream.

The algorithm is so straightforward that it does not require a key to be implemented. Whilst this makes things a lot simpler to program and exchange the secret, it does mean that the security lies solely in the algorithm. If a key were used, then it might still be impossible for the adversary to decode the hidden message, as the key would usually index the manipulated regions of the image. In the case of the *Hide & Seek* algorithm however, the adversary simply needs to understand how the algorithm works, and they will be able to decrypt the message.

Encoding

Algorithm 1 The encoding process of the *Hide & Seek* algorithm in *sequential* mode.

```
1: for i = 1, ..., l(m) do
2:     p ← LSB(c_i)
3:     if p ≠ m_i then
4:         c_i ← m_i
5:     end if
6: end for
```

The encoding process (as shown by the pseudocode in Algorithm 1) shows that the entire algorithm can be implemented by writing just a few lines of code. The algorithm works by taking the first pixel of the image c_i and obtaining its LSB value (as per line 2 of the Algorithm). This is typically achieved by calculating the modulus 2 of the pixel value. This will return a 0 if the number is even, and a 1 if the number is odd, which effectively tells us the LSB value. We then compare this value with the message bit m_i

that we are trying to embed. If they are already the same, then we do nothing, but if they are different then we replace c_i with m_i. This process continues whilst there are still values in m that need to be encoded.

Decoding

The decoding phase is even simpler. As the encoder replaced the LSBs of the pixel values in c in sequence, we already know the order that should be used to retrieve the data. Therefore all we need to do is calculate the modulus 2 of all the pixel values in the stegogramme s, and we are able to reconstruct m as m'_i. Algorithm 2 shows the pseudocode of the decoding process.

Algorithm 2 The decoding process of the *Hide & Seek* algorithm in *sequential* mode.

1: **for** $i = 1, ..., l(s)$ **do**
2: $\quad m'_i \leftarrow \text{LSB}(s_i)$
3: **end for**

Note that this time we run the loop for $l(s)$ instead of $l(m)$. This is because the decoding process is completely separate from the encoding process and therefore has no means of knowing the $l(m)$. If a key were used, it would probably reveal this information, but instead we simply retrieve the LSB value of every pixel. When we convert this to ASCII, the message will be readable up to the point that the message was encoded, and will then appear as gibberish when we are reading the LSBs of the image data.

3.2.3 Hide & Seek: The Randomised Approach

The *randomised* approach to the *Hide & Seek* algorithm makes it possible to scatter the locations of the pixels that are to be replaced with the message data. The core of the encoding process is identical to that of the sequential method described in section 3.2.2. In fact, the two methods only differ in terms of how the image data c_i is presented **before** the embedding process starts.

For the randomised approach the image data c is usually shuffled using a Pseudo Random Number Generator (PRNG). This generator will take the image data c and produce a shuffled version C according to a seed k that is specified by the encoder. There will also be an inverse shuffle which takes C and returns the original order c when the same k is used. The pixel values of the image c are often shuffled before embedding such that the exact same encoding mechanism from section 3.2.2 can be used. The values are then shuffled back to their original positions after embedding such that the image can be displayed properly for sending it across some communications channel to the recipient. A PRNG also has the advantage that it produces the same shuffle when the same data and the same seed are given back to it. This means that all we need is c and k at the decoding stage, and the same shuffle will be recreated so we can retrieve the message data successfully.

Encoding

Algorithm 3 shows the pseudocode for the encoding process of the *randomised Hide & Seek* approach. Note how the bulk of the encoding process remains the same as for the sequential embedding approach. However, we now have line 1 that randomises the locations of each pixel before embedding the message data. In addition to this, we also have line 8 which returns the pixel locations back to normal when the embedding process has ended. The seed k acts as a key to the algorithm such that the same shuffle sequence can be generated when retrieving the hidden message.

The output stegogramme s from this embedding approach will contain bits of the hidden message in seemingly random locations of the image.

Algorithm 3 The encoding process of the *Hide & Seek* algorithm in *randomised* mode.

```
1: generate randomised sequence C using data c and seed k
2: for i = 1, ..., l(m) do
3:     p ← LSB(C_i)
4:     if p ≠ m_i then
5:         c_i ← m_i
6:     end if
7: end for
8: generate original sequence c using data C and seed k
```

Perhaps the most important aspect of note is that as we require k to identify the correct regions, the algorithm is much more secure than the sequential approach, as the sequence cannot be derived without it.

Decoding

By using the seed k that was created in the encoding process, the pixel values from the stegogramme s can be shuffled into the same sequence that they were when embedding the secret message. From here, the same decoding process as described in section 3.2.2 can be used to obtain each m_i, and thus recreate the complete message stream m. The decoding process can be seen in Algorithm 4.

Algorithm 4 The decoding process of the *Hide & Seek* algorithm in *randomised* mode.

```
1: generate randomised sequence S using data s and seed k
2: for i = 1, ..., l(s) do
3:     m'_i ← LSB(S_i)
4: end for
```

Sometimes, as a seed is already required to retrieve the message, the randomised approaches may go one step further and create a full key that also declares $l(m)$. If this is the case, line 2 can be changed such that the loop runs for $l(m)$ rather than $l(s)$.

3.3 Transform Domain Techniques

As we will see in Chapter 4, embedding the message data directly into the spatial domain (as per the *Hide & Seek* method) means it is quite straightforward to detect that embedding has taken place. To counteract this, new methods were developed that embedded the message data in more inconspicuous areas - the most popular being the *transform domain*.

3.3.1 Overview

When JPEG images are compressed to smaller file sizes, they are firstly converted into the *Discrete Cosine Transform (DCT)* domain which presents the data as *high* and *low* frequencies. High frequencies relate to areas of high detail, and low frequencies are the low detailed areas. The logic behind the JPEG compression process is that we can remove some of the high detail because our eyes are less sensitive in these areas, meaning we would not notice if some of it was not there. This logic is based on the fact that our eyes are most sensitive in the plainer regions of an image. To illustrate this by example, think of an image of a very dense forest. If you change a few random pixels to be black, the chances are you will not be able to see them because your eyes will be so distracted by all the other detail. However, if the picture was of a plain-painted wall, you may have been able to see those black pixels more clearly because there is less of a distraction (unless of course, the wall was also painted black).

As the DCT values (referred to as *coefficients*) are tweaked when compressing, we can similarly tweak some of the values such that they hold message data. Embedding in this fashion is much harder to notice from a steganalytical viewpoint than embedding in the spatial domain, as the steganalyst will have to do a bit more digging to find any artifacts of embedding. There are several transforms that could potentially be used to embed the hidden data, including the *Discrete Wavelet Transform (DWT)*, *Fast Fourier Transform (FFT)*, *Walsh-Hadamard Transform*, and many more. However, to keep things at a comprehendible level, we will only discuss the methods used for embedding in the DCT domain .

3.3.2 JSteg

The *JSteg* algorithm was developed by Derek Upham and is esentially a carbon copy of the *Hide & Seek* algorithm discussed in section 3.2.2, because it employs sequential least significant bit embedding. In fact, the *JSteg* algorithm only differs from the *Hide & Seek* algorithm because it embeds the message data within the LSBs of the DCT coefficients of c, rather than its pixel values.

Before the embedding process begins, the image is converted to the DCT domain in 8x8 blocks such that the values of c_i switch from pixel values to DCT coefficients. In order for the values to be presented as whole numbers, each 8x8 block is *quantised* according to a *Quantisation Table Q*. The result is where the embedding algorithm operates. An example of an 8x8 DCT block is shown in Figure 3.4.

257	6	3	0	0	0	-6	7
8	0	1	-5	2	4	-4	3
-5	-1	-1	1	-1	-1	2	-2
2	2	2	2	-1	-2	0	0
-1	-2	0	-2	2	1	-1	0
2	-1	-2	0	-2	1	2	1
-2	0	3	2	2	-3	-1	-1
2	0	-2	-2	-1	2	0	1

Figure 3.4: An example of an 8x8 sub-block of DCT coefficients.

We should also note the two types of coefficient that we see in every 8x8 block: *DC*, and *AC*. The value at the top left of each 8x8 block is known as the *DC coefficient*. It contains the mean value of all the other coefficients in the block, referred to as the *AC coefficients*. The DC coefficients are highly important to each block as they give a good estimate as to the level of detail in the block. Changing the value of the DC coefficient will also change many of the values of the AC coefficients, and this will create a visual discrepancy when the image is converted back to the spatial domain and viewed normally. For this reason, the *JSteg* algorithm does not embed message data over any of the DC coefficients for every block. In addition to this, the algorithm also does not permit embedding on any AC coefficient equal to 0 or 1.

Encoding

Algorithm 5 provides the pseudocode for the encoding process of the *JSteg* algorithm. Line 4 shows that the algorithm avoids embedding on the DC coefficients, and also any AC coefficient equal to 0 or 1. Line 8 shows an alternative method for calculating the LSB value of the coefficient by using mod 2. The result is replaced with the value in m_i.

Again, no key is used for this algorithm. So long as the decoder knows that the embedding took place in the DCT domain, it will be capable of extracting the message successfully. The security of the *JSteg* algorithm therefore lies in the algorithm itself. As we noted before, the main difficulty of not using a key is when we try to determine $l(s)$ when extracting the message. Without a key, it is impossible to know the length of the message to extract, so the loop is typically run for the entire duration of the image to ensure that all of the message is extracted. This is certainly the case for the *JSteg* algorithm as we will see in the decoding process.

Algorithm 5 The encoding process of the *JSteg* algorithm.

1: convert image c to DCT domain d in 8x8 blocks
2: **for** $i = 1, ..., l(m)$ **do**
3: $p \leftarrow \text{DCT}(d_i)$
4: **while** p = DC or p = 0 or p = 1 **do**
5: p = next DCT coefficient from d
6: **end while**
7: $p_i \leftarrow c_i \bmod 2 + m_i$
8: $c_i \leftarrow p_i$
9: **end for**
10: convert each 8x8 block back to spatial domain

Decoding

The decoding process functions by converting the stegogramme s to the DCT domain. It then avoids the same coefficient values that the encoding algorithm avoids, and retrieves the hidden message from the LSBs of all the other coefficients sequentially (line 7).

Algorithm 6 The decoding process of the *JSteg* algorithm.

1: convert image s to DCT domain d in 8x8 blocks
2: **for** $i = 1, ..., l(s)$ **do**
3: $p \leftarrow \text{DCT}(d_i)$
4: **while** p = DC or p = 0 or p = 1 **do**
5: p = next DCT coefficient from d
6: **end while**
7: $m_i \leftarrow d_i \bmod 2$
8: **end for**

3.3.3 OutGuess 0.1

In much the same way that embedding the message data sequentially using the *Hide & Seek* method was not considered very secure, neither was the fact that the *JSteg* algorithm embedded in the same fashion. The first version of *OutGuess*, designed by Neils Provos [18], improved the *JSteg* algorithm by scattering the embedding locations over the entire image according to a PRNG on image c derived using seed k. This is very similar to the way that the *randomised* embedding approach improved the *Hide & Seek* algorithm.

Encoding

The encoding process of the *OutGuess 0.1* algorithm is essentially a combination of both the *randomised Hide & Seek* algorithm and the *JSteg* algorithm. As Algorithm 7 shows, the first step is to convert the image to the DCT domain, in exactly the same way as we

saw for *JSteg*. Then, the coefficients are shuffled into a seemingly random order using a PRNG according to a seed. The message data is then embedding using the same technique as for *JSteg* before finally inversing the shuffle such that the coefficients are back in the correct positions. The image is then converted back to the spatial domain and thus the stegogramme s is produced.

The algorithm still avoids embedding within the DC coefficient, and any AC coefficient equal to either 1 or 0. This is exactly the same methodology that we saw with the *JSteg* algorithm.

Algorithm 7 The encoding process of the *OutGuess 0.1* algorithm.

1: convert image c to DCT domain d in 8x8 blocks
2: generate randomised sequence C using data d and seed k
3: **for** $i = 1, ..., l(m)$ **do**
4: $\quad p \leftarrow \text{DCT}(C_i)$
5: $\quad$ **while** p = DC or p = 0 or p = 1 **do**
6: $\quad\quad$ p = next DCT coefficient from C
7: $\quad$ **end while**
8: $\quad p_i \leftarrow C_i \bmod 2 + m_i$
9: $\quad C_i \leftarrow p_i$
10: **end for**
11: generate original sequence c using data C and seed k
12: convert each 8x8 block back to spatial domain

Decoding

The decoding process for *OutGuess 0.1* is as we might expect. Firstly, the stegogramme s is converted to the DCT domain, before being shuffled using the same k that was used in the encoding process. We then retrieve the message data by extracting the LSBs from all the coefficients whose values are neither a DC value, nor a 0 or a 1. Algorithm 8 shows the pseudocode for the decoding process of the *OutGuess 0.1* algorithm.

Algorithm 8 The decoding process of the *OutGuess 0.1* algorithm.

1: convert image s to DCT domain d in 8x8 blocks
2: generate randomised sequence S using data d and seed k
3: **for** $i = 1, ..., l(s)$ **do**
4: $\quad p \leftarrow \text{DCT}(S_i)$
5: $\quad$ **while** p = DC or p = 0 or p = 1 **do**
6: $\quad\quad$ p = next DCT coefficient from s
7: $\quad$ **end while**
8: $\quad m_i \leftarrow s_i \bmod 2$
9: **end for**

3.3.4 OutGuess 0.2

When the *OutGuess 0.1* algorithm was developed, it was considered much more secure than the *Hide & Seek* and *JSteg* algorithms because it not only embedded the information in a more discrete area of the image (DCT coefficients), it also scattered the locations of embedding by using a PRNG to shuffle the ordering of the coefficients.

However, after the release of the algorithm, steganalysts were able to find a fatal flaw in the technique that left statistical artifacts in the stegogrammes. As a result, Neils Provos [19] created a revised version of the *OutGuess 0.1* algorithm, called *OutGuess 0.2*. It was written in an attempt to ensure that the statistical properties of the cover image c were maintained after embedding, such that stegogramme s looks statistically similar to a clean image. This would make it harder for steganalysts to calculate the likelihood that their suspect image is a stegogramme.

The actual embedding algorithm is exactly the same for *OutGuess 0.2* as it was for *OutGuess 0.1*. The difference lies in what happens after the information has been embedded. In *OutGuess 0.2*, corrections are made to the coefficients such that they appear similar to that of a clean image in terms of frequencies of the values. This is known as *statistics-aware* steganography. The impact of these corrections will become much more apparent when we take a look at statistical steganalysis in Chapter 4.

3.3.5 F3

As an alternative to the *OutGuess 0.2* algorithm, Andreas Westfeld designed an algorithm called *F3* [27] which was considered even more secure. The reason for this is that it did not instantiate the same embedding process as the *JSteg* and *OutGuess* algorithms. Instead of avoiding embedding in DCT coefficients equal to 1, the *F3* algorithm permitted embedding in these regions, whilst it would still avoid embedding in zeros and the DC coefficients. The algorithm still embedded the message data sequentially within c.

Another change with this algorithm was that it did not embed directly in the least significant bits of the DCT coefficients, but instead took the absolute value of the coefficients first, before comparing them to the message bits. If both the absolute value of the coefficient, and the message bit were the same, then no changes are made. If they are different, then the absolute value of the DCT coefficient is reduced by 1. An implication of this however, is that zero values are often created which the decoding algorithm will not be programmed to extract data from. The *F5* algorithm worked around this by re-embedding m_i when the result is that a zero DCT coefficient is created. This re-embedding was referred to as *shrinkage*.

3.3.6 Encoding

Algorithm 9 shows the pseudocode for the encoding process of the *F3* algorithm. The key part of the function can be seen on line 7. It shows how the absolute value of the current coefficient is taken such that it can then be compared with the message bit m_i. If they are both the same then we can move on to the next d_i coefficient and attempt to

embed the next m_i value. When $P \neq m_i$, then the absolute value of P is reduced by 1 and inserted into d_i.

The *shrinkage* handling process is shown in the loop beginning on line 15. This shows that if a zero value is created, then the same m_i value is embedded in the next iteration.

Algorithm 9 The encoding process of the *F3* algorithm.

```
1:  convert image c to DCT domain d in 8x8 blocks
2:  for i = 1, ..., l(m) do
3:    p ← d_i
4:    while p = DC or p = 0 do
5:      p = next DCT coefficient from d
6:    end while
7:    P ← absolute(p_i)
8:    if P ≠ m_i then
9:      P ← P − 1
10:     absolute(d_i) ← P
11:   end if
12:   if d_i = 0 then
13:     next m_i = m_i
14:   end if
15:   C_i ← p_i
16: end for
17: convert each 8x8 block back to spatial domain
```

Note also that the constraints for this algorithm (line 4) are different to those of the algorithms we have previously seen. There is nothing to suggest that embedding on coefficient values equal to 1 is not permitted.

3.3.7 Decoding

The decoding process for *F3* is slightly less complicated that the encoding process because it does not concern the *shrinkage* issue. As Algorithm 10 shows, if the coefficient value is equal to zero, it does not attempt to retrieve m_i. This is illustrated in line 4.

Algorithm 10 The decoding process of the *F3* algorithm.

```
1: convert image s to DCT domain d in 8x8 blocks
2: for i = 1, ..., l(m) do
3:   p ← d_i
4:   while p = DC or p = 0 do
5:     p = next DCT coefficient from d
6:   end while
7:   P ← absolute(p_i)
8:   m_i = P
9: end for
```

3.3.8 F4

The main pitfall with *F3* was the fact that it effectively embedded more zeros than ones as a result of the *shrinkage* mechanism. This meant that when the statistical properties of the stegogramme s are examined through its histogram for example (we will come back to this in Chapter 4), some artifacts of embedding became apparent. This is much the same as what happened in the *JSteg* implementation except a slightly different pattern is derived.

In addition to this, steganalysts also found that more odd coefficients existed in *F3* stegogrammes than even coefficients. This now meant that there were **two** deficiencies that could be examined when viewing the histogram of a suspect image. F4 was developed to remove these properties such that the histogram would appear similar to that of a clean image.

Encoding

The *F4* algorithm eliminates the two weaknesses of *F3* in one stroke by mapping negative coefficients to the steganographic value, where even-negative coefficients = steganographic 1, odd-negative coefficients = 0, even-positive coefficients = 0 (as with *JSteg* and *F3*), and, odd-positive coefficients = 1 [27]. Put more simply, this means that now, if we embed a 0 in a DCT coefficient equal to -3, the result will remain -3, where as it would have been modified to -2 using *F3*. This means that the bit-flips now occur with roughly the same probability, so the histogram for the stegogramme will not appear unstructured in terms of its frequency distribution.

Algorithm 11 shows a simplified approach to the encoding process of the *F4* algorithm. Note how the coefficients are evaluated in terms of being either positive or negative in lines 8-11 respectively. They are then either incremented or decremented according to their existing state.

Decoding

The decoding process operates as we would expect. The stegogramme s is converted to obtain the quantised DCT coefficients. From here we ensure that we skip the DC values and any value equal to zero. However, all other values are used to retrieve the message data in accordance with the encoding algorithm.

Algorithm 12 displays the pseudocode for the decoding phase of the *F4* algorithm. Note how we subtract 1 from P in line 9. This is because the encoding algorithm added 1 at this stage when embedding. So to ensure we yield the correct bit for m_i, we must reflect this change. Similarly, the value in line 11 is incremented by 1 because it was decremented at this same stage in the encoding process.

Algorithm 11 The encoding process of the *F4* algorithm (simplified) [27]).

```
1: convert image c to DCT domain d in 8x8 blocks
2: for i = 1, ..., l(m) do
3:   p ← d_i
4:   while p = DC or p = 0 do
5:     p = next DCT coefficient from d
6:   end while
7:   P ← absolute(p_i)
8:   if P = m_i and P > 0 then
9:     P ← P + 1
10:    absolute(d_i) ← P
11:  else if P ≠ m_i and P < 0 then
12:    P ← P − 1
13:    absolute(d_i) ← P
14:  end if
15:  if d_i = 0 then
16:    next m_i = m_i
17:  end if
18:  C_i ← p_i
19: end for
20: convert each 8x8 block back to spatial domain
```

Algorithm 12 The decoding process of the *F4* algorithm.

```
1: convert image s to DCT domain d in 8x8 blocks
2: for i = 1, ..., l(m) do
3:   p ← d_i
4:   while p = DC or p = 0 do
5:     p = next DCT coefficient from d
6:   end while
7:   P ← absolute(p_i)
8:   if P = m_i and P > 0 then
9:     m_i ← absolute(p_i) − 1
10:  else if P ≠ m_i and P < 0 then
11:    m_i ← absolute(p_i) + 1
12:  end if
13: end for
```

3.3.9 F5

The *F5* algorithm [27] is predominantly the same as the *F4* algorithm, at least in terms of its strategy for encoding the message data. However, the *F5* algorithm was designed in an attempt to improve on the *F4* algorithm by minimising the disturbance caused on c when embedding the message data. This was achieved by introducing *matrix encoding*, and the

algorithm was the first known stego-system to make use of this technique. We will not review matrix encoding in great detail as it is rarely used for steganography, however we should be aware that it significantly decreases the necessary number of changes required for embedding the message data.

The algorithm itself works by calculating the embedding potential of encoding m within c based on m_i. Hamming coding is then used to embed potentially more than one bit per value by making no more than 1 change to the coefficients. This is denoted as $2^p - 1$, where p refers to the number of bits that are to be embedded [14].

The *F5* algorithm is therefore much more secure than the *F3* and *F4* algorithms, because there is less of a difference between c and s after embedding. Of course, the longer $l(m)$ is, the more changes need to be made to produce s. However, there are notably few modifications than we would see when using the other encoding algorithms.

3.4 Conclusion

The stego-systems described in this section have presented some of the most significant algorithms developed for image steganography. They demonstrate just a handful of techniques that make it possible to encode message data.

As we have seen, the level of complexity increases for each new algorithm. This is to be expected as one algorithm attempts to improve on the version that existed before it, reviewing in particular how steganalysts managed to break the older systems. From this information, the algorithm is designed such that it confuses the same steganalytical attack into believing that a stegogramme is a clean image. In other words, it attempts to increase the amount of *false-negatives* that pass through a steganalytical filter, where the term *false-negatives* refers to cases where the steganalysis attack considers the suspect image to be clean.

We have also seen that some stego-systems use a key to randomise the positions of the manipulated values. Keyed steganography is often preferred as it creates an inconsistency when steganalysts review its stegogrammes. It will typically be a harder task of identifying steganography in random locations, than doing so for sequentially embedded implementations. However, it is not secure enough just to randomly distribute the embedding locations; the algorithm needs to ensure that the domain of the embedding locations themselves are not glaringly obvious. More up to date methods of steganography do not even randomise the embedding locations themselves, but rather they randomise the encoding algorithm according to the value in each location. YASS (Yet Another Steganographic Scheme) is a good example of such an implementation.

All of the above approaches can be adapted to work for both grayscale (8-bit) and truecolour[2] (24-bit) images. Typically, there is more redundancy in a truecolour image because it has 3 times as many layers as that of a grayscale image. However, steganalytical approaches have been developed for detecting steganography from within any type of

[2]The term *truecolour* refers to images that contain three colour planes of information (red, green, and blue), each represented by 8-bits. Each pixel contains colour intensity values for each plane, resulting in 24-bits (or 3 bytes) worth of information.

image, so it should not make too much of a difference choosing one image type over the other.

What is also worth noting is that a good method of practise is to keep the message data as short as possible when using steganography. The more data there is, the more disturbance is caused to the image, whereas a shorter message would yield fewer modifications. Therefore, a stegogramme containing a high capacity message is traditionally a higher risk than one with a shorter capacity message, even though they have encoded the message data in the same way. It took some time for steganographer's to really understand the importance of reducing the length of the message data, but *F5* shows how serious an issue it is, as it incorporates matrix encoding to keep the disturbances to a minimum.

4 Targeted Steganalysis

Chapter 3 served both as an illustration of just a few steganographic algorithms that can be used to embed a secret message within an image, and also as an indication of the simplicity of implementing these algorithms. If we are to assume that steganography is used with ill-intent (such as terrorism) then it is imperative that we continually develop steganalytical schemes capable of breaking steganography.

At the very least, the schemes must be capable of identifying the existence of a hidden message with high confidence, whilst the aspiration of recovering the message should be considered extremely unlikely due to fact that cyptographically secure random number generators are often employed to scramble the message when embedding, thus obscuring the integrity of the message. For this reason, data recovery is almost always ignored by steganalysts.

This chapter presents some basic steganalytical schemes associated with "targeted" steganalysis, including *visual*, *structural*, and *statistical* attacks. We will focus on demonstrating how they can be used to combat the steganographic algorithms discussed in Chapter 3, as well as introducing techniques targeted for previously unmentioned stego-systems. Finally, the chapter will present the strengths and weaknesses of the techniques in a concluding summary.

4.1 Visual Attacks

Visual Attacks are widely regarded as the simplest form of steganalysis. As the name suggests, a visual attack largely involves examining the subject file with the naked eye to identify any obvious inconsistencies. Of course, the first rule of steganography is that any modifications made to a file should not result in quality degradation, so a good steganographic implementation will create stegogrammes that do not look any more suspicious than the cover Work - at least not at face value.

However, when we remove the parts of the image that were not altered as a result of embedding a message, and instead concentrate on the likely areas of embedding in isolation, it is usually possible to observe signs of manipulation. It can therefore be argued that the key aspect of a successful visual attack is to correctly determine which features of the image can be ignored (*redundant* data), and which features should be considered (*test* data) in order to to test the hypothesis that a suspect image contains steganography. An incorrect choice can lead to an increase in false-negatives, which is something a steganalyst would want to avoid. As a result, it is highly likely that all permutations of possible *redundant* and *test* data sets will be analysed such that the steganalyst is in the strongest position to make an informed conclusion.

The most common form of a visual attack combats LSB steganography (such as the *Hide & Seek* algorithm) and is made possible by the fact that the structure of the LSBs for an image does not match the structure of message bits (if converted to binary from ASCII[3] characters). For an image, there are typically as many even values as there are odd, which is equal to saying that there are typically as many 1's as there are 0's in its LSB plane. When text is converted to binary however, there are often more 0's than 1's, and this produces a visual inconsistency when such a bit stream resides alongside the image's LSB plane. This inconsistency is what the steganalyst will be looking to find in order to classify a suspect image as either an innocent image or a stegogramme.

Of course, how clearly we notice these inconsistencies depends on how good (or how bad) the steganographic strategy is. A poor strategy will embed the message bits directly after converting from ASCII to binary, and this will lead to the increase in 0's. However, as we will see in section 4.1.3, if the bit stream is encrypted, then there will usually be a more even distribution of 1's and 0's. This means that the discrepancies will not stand out so clearly.

4.1.1 Visual Attacks on Sequential Hide & Seek

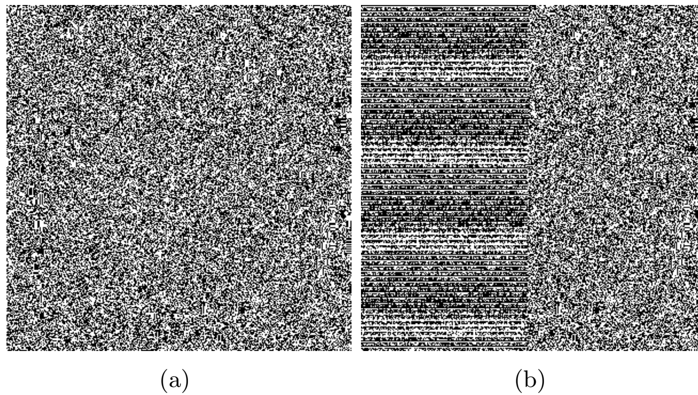
The earliest *Hide & Seek* implementation functioned by replacing the LSBs of the image with the bit stream of the message in sequence until all of the message had been embedded. When constructing a visual attack to identify whether or not a suspect image has been subjected to this kind of embedding, the steganalyst will be looking to obtain a visual discrepancy for the first k pixels where k is the length of the message. It is worth nothing however, that the exact value for k will not be made apparent until the visual attack successfully indicates signs of embedding.

Figure 4.1 illustrates an example of a visual attack on a grayscale BMP image. It shows what the LSB plane looks like for (a) a clean image, and (b) an image that is suspected to have been manipulated using *Hide & Seek* in sequential pixels of the image.

It is clear that image (b) contains hard evidence of tampering as the first ~50% of the LSBs do not look as unorganised as the latter half of the image. This is a strong indication that the suspect image is a stegogramme. On top of this, it is also possible to derive an estimate of the message length to be k = ~50%. We could even calculate the message length more precisely by dividing the total number of modified pixels by the total number of pixels.

Note also that whilst the example in Figure 4.1 shows the LSB plane of the clean image, it is not a required input for the attack to function. Without the clean image the steganalyst would still derive image (b) from implementing the attack, which would be adequate for concluding that the image is a stegogramme.

[3]American Standard Code for Information Interchange: character encoding based on letters of the English alphabet, http://www.asciitable.com/

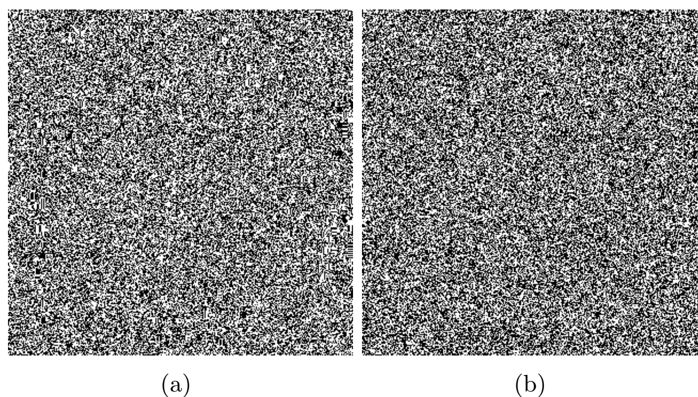


(a) (b)

Figure 4.1: LSB planes of (a) a clean image, and (b) a suspect image.

4.1.2 Visual Attacks on Randomised Hide & Seek

It is much harder to perform a visual attack with any success when a stegogramme was created by embedding into seemingly random locations of LSB values. Where as for sequential embedding we effectively saw a non-random bit stream overlapped over an otherwise random bit plane, the non-random bit stream is now split up and dotted about all over the image. When we look at the resulting LSB plane in isolation, it is much more difficult to identify the regions that have been altered as a result of embedding (especially if the embedded message is very short).



(a) (b)

Figure 4.2: LSB planes of (a) a clean image, and (b) a suspect image.

Figure 4.2 shows the LSB planes of (a) a clean image (zero embedding), and (b) the corresponding LSB plane of a stegogramme that was created by randomised LSB embedding. If we compare image (b) with the respective image from section 4.1.1, it is reasonable to conclude that visual attacks do not highlight the manipulated regions as

clearly when the embedding locations are randomised. If we assume - as is most often the case - that the steganalyst does not have access to the clean image, then it will be a near impossible task to determine whether or not the image contains steganography as there are no obvious artifacts that point to signs of embedding. This was not the case with image (b) in section 4.1.1, where the manipulated regions were clearly highlighted in the first ~50% of the image, and so a firm analysis could still be made even when the steganalyst is not able to compare the LSB plane of the suspect image against that of the clean image.

Visual attacks on random embedding therefore differ from those for sequential embedding in that the approach relies on the steganalyst having access to the clean image in order to prove successful; this is referred to as a *known cover* attack. When the clean image is available, the steganalyst can obtain its LSB plane as well as the LSB plane of the suspect file, and then calculate the difference between the two by subtracting one from the other. They will then be left with an image similar to that in Figure 4.3 where the black regions of the image represent the values that do not change between both images, and the white regions represent the areas that have changed.

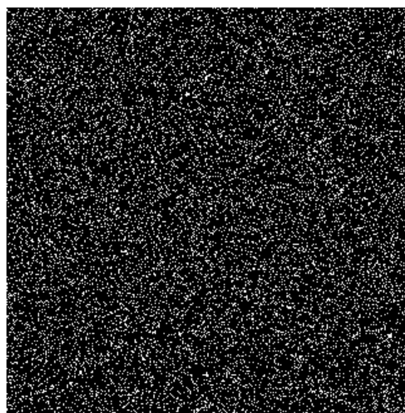


Figure 4.3: The difference between the LSB planes of the original and stego images.

If the image in Figure 4.3 were completely black, it would mean that there is no difference between both LSB planes. However, as there are white regions within the image, we can see that there are some differences between the LSB planes of the suspect file and the clean image. The steganalyst can therefore conclude that the suspect image has been altered, and they can even identify the locations of the modified pixels, as well as deriving an estimate for k just as it was possible in section 4.1.1. In this instance, it again appears that $k =$ ~50%.

4.1.3 Summary

A successful visual attack not only allows the steganalyst to pin-point inconsistencies within an image, it also divulges how the stego-system operated (i.e. sequential/randomised embedding in the spatial LSBs of the image), and allows for an esti-

mation of the message length. When we consider these benefits along with how simple the attack is to carry out, we can conclude that visual attacks can be a useful tool for steganalysis - at least for detecting steganography from the most basic steganographic implementations, or for *known cover* attacks.

However, for all instances where the clean version of the image is not available to the steganalyst, we are dependant on three factors holding true in order for visual attacks to prove successful. The message must:

1. be embedded in a sequential order
2. be of a length less than the maximum size of the bit plane
3. not be encrypted, but instead converted to binary from ASCII characters

Only the most naïve stego-systems apply all of these factors, and finding a system that does not encrypt a message before embedding is particularly challenging. Encrypting the secret message is a fairly simple process, and if implemented properly it can reduce the chance of success for a visual attack by considerable proportions (assuming the clean image is not available).

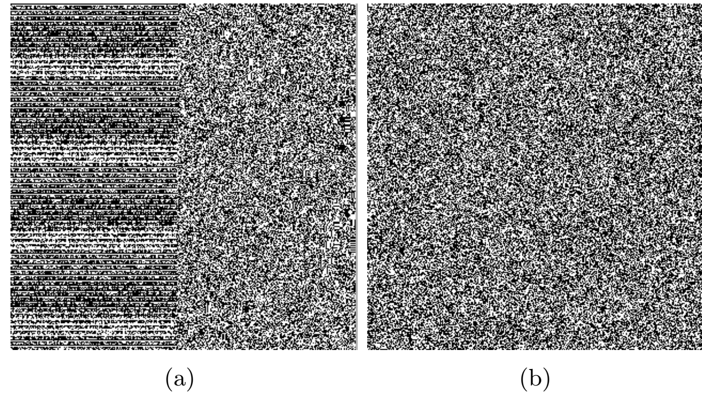
The underlying reason for the success of the attack in section 4.1.1, is due to the fact that in this instance the message was embedded directly after it was converted from ASCII to 8-bit binary sequences. When converting and embedding in this manner, we create an unintential bias towards the frequency of zeros in the embedding data, and this is why we see the artifacts of embedding with such profound strength. The bias is caused because of the structure of the binary sequences that are created after conversion. Uppercase and lowercase English letters fall in the ranges 65-90, and 97-122 respectively according to the ASCII table, and these tend to be the most frequently used characters for communicating a secret message. When we convert numbers within these ranges to 8-bit binary (assuming we are catering for the full 256 "extended ASCII" characters), we obtain sequences such as:

$$\begin{aligned} 65 &= 01000001 & 97 &= 01100001 \\ 66 &= 01000010 & 98 &= 01100010 \\ 67 &= 01000011 & 99 &= 01100011 \\ 68 &= 01000100 & 100 &= 01100100 \end{aligned}$$

What we witness here is the fact that the frequency of 0's dramatically outweighs the frequency of 1's, and this holds true for a conversion of all alphabet characters. Of course, when we sequentially replace the seemingly even distribution of LSB values of an image with a magnitude of 0's, and relatively few 1's, a pattern emerges that the human eye is capable of detecting quickly. If we assume that the message is shorter than the full embedding potential, the pattern is further emphasised by a change in form when the message bits end, and the LSBs return to representing the image data.

Figure 4.4 highlights the significance of encrypting a message before it is embedded. Image (a) shows what is seen by the steganalyst if the message is embedded sequentially, and directly after converting to ASCII. Simply by encrypting the message before embedding, the steganalyst will be presented with image (b) when the same test is run. Clearly,

it is much harder to see that image (b) contains signs of embedding, so the visual attack becomes less useful.



(a) (b)

Figure 4.4: The significance of encrypting the message: (a) embedding directly after converting to binary from ASCII, (b) embedding by encrypting the message data before embedding.

Let us accept however, that there will be circumstances where messages are left unencrypted, therefore meaning that steganalysts are able to view a visual discrepancy within a suspect image. In order to view the discrepancies clearly, they rely on the fact that the message length is shorter than the full embedding capacity of that bit plane. If the message were embedded within all LSBs of the image, then the steganalyst would be left with an image close to that of Figure 4.5. It is no longer possible to see a change in form in the bit plane, so the steganalyst will find it harder to place as much confidence on the fact that the image might be a stegogramme.

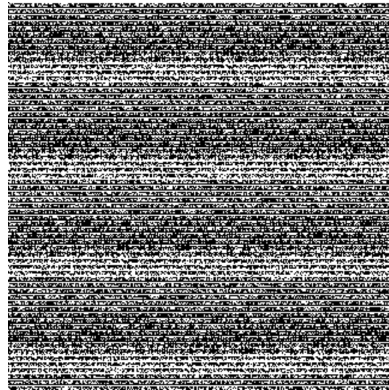


Figure 4.5: The LSB plane for a stegogramme containing a message at full capacity.

Of course, an evaluation could still be made as a result of what the LSB plane of the suspect image looks like in comparison with the content of the entire image. If the image

is a photograph of a person, then it is highly unlikely that the LSB plane should look like that presented in Figure 4.5. If the image is of a series of horizontal lines however, then this structure of LSB plane is much less radical.

We have also seen that visual attacks are not as likely to succeed for any steganographic implementation that randomises the positions of the affected LSBs. If the steganalyst has the luxury of having access to the cover Work, it is possible to plot the difference between its bit plane and that of the suspect image. However, for the most part, the cover Work will not be available, in which case the steganalyst has to rely on what they see in the suspect image's bit plane alone.

4.1.4 Further Development

One of the traits that makes a visual attack so different from other steganalytical techniques is the fact that the decision-making process is not automated. Whilst a computer may be used to strip an image of the redundant data and display only the test data, the act of classifying whether or not the image is a stegogramme is open to personal interpretation as opposed to solid statistics. This can either be interpreted as a strength or a weakness; a strength because there is no reliance on theoretical expertise, but a weakness because people's opinions may vary, often resulting in inconsistent interpretations of what constitutes a clean image or a stegogramme.

However, with research by Wayner [24], and Westfeld [26] suggesting that every bit plane - including the LSB plane - is non-random, perhaps it could be possible to broaden these findings, and develop a more structured and logical approach to the decision-making process. If we can consider each bit plane of an image P_i to be non-random, then a plausible method must exist for determining the expected values that make up a bit plane for a suspect image $E(P_i)$. It is likely to be possible by using the data from the unmodified bit planes, such that we can fill in the gaps for the bit plane in consideration. It should then be a relatively trivial task to spot arbitrary modifications to any bit plane when analysing the suspicious bit plane $\hat{P}_i$ against $E(P_i)$. There are two possibilities:

$$E(P_i) = \hat{P}_i \tag{4.1}$$

$$E(P_i) \neq \hat{P}_i \tag{4.2}$$

If equation (4.1) is true, then there is no difference between the bit plane of the suspect image, and what we expect the plane to look like. Thus we can conclude that the image has not been altered, and it is unlikely to be a stegogramme. If equation (4.2) is true however, then the two bit planes are inconsistent, and we can assume that some modifications have taken place. In this instance, the image can be considered a likely stegogramme.

If we can estimate each bit plane for an image, then visual attacks will become a lot more desireable to steganalysts as there will be less dependance on use for a known-cover attack. Instead, we can predict the clean image and identify the likelihood of embedding based on the differences that are highlighted between properties of our estimate and the suspect image.

4.1.5 Conclusion

Visual Attacks can potentially be implemented in many ways, picking on different properties of the image in each implementation such that a wide range of embedding strategies are examined. For example, a visual attack could be set up to display the spatial LSB plane of the image on its own, the two lowest bit planes, the second lowest bit plane only, and so on. When all possible bit plane combinations have been exhausted, the steganalyst will be better placed to determine whether or not the image was subjected to spatial steganographic embedding such as that defined by the Hide & Seek [19] method (or a slight modification of that algorithm that embeds in a different bit plane or combination of bit planes). Similarly, the steganalyst could also exhaust all possible bit plane combinations in a particular transform (i.e. the Discrete Cosine Transform) to evaluate whether or not the image contains signs of transform embedding such as those expected by an implementation of the JSteg [19] method. In addition to this, it is also possible to target the attack against a variety of image formats, (such as JPEG, GIF, BMP, PNG etc.) which only helps to make the concept more useful as an investigative tool as it can test against many thousands of possible embedding approaches. It is worth noting though that the success of the attack will vary significantly depending on the type of steganography used, and the format of the resulting image. The fact that the attack can be applied in many different ways illustrates a commendable diversity that other algorithms do not offer, however, whilst we may examine different properties of several image formats, we still require either the clean image, or a naïve steganographic implementation in order to expose the damaged regions well and achieve optimal success.

Perhaps the biggest pitfall with the attack is the fact that it cannot be automated. It can therefore prove very time-consuming to produce test images for several possible methods of embedding, and that is before they are perceptually analysed. If a steganalyst wishes to exhaust every type of embedding strategy, they would need to look at thousands of images to consider the likelihood that a single suspect image is a stegogramme. This is obviously an inefficient methodology, and is often the reason why other steganalytical methods are preferred.

4.2 Structural Attacks

Structural attacks are designed to take advantage of the high-level properties that are known to exist for a particular steganographic algorithm. For example, version 4.1 of *Hide & Seek* was forced to operate only on images that were of size 320 x 480 pixels [19]. Similarly, *StegoDos* operated only on images of size 320 x 200 pixels [10]. This means that a steganalyst that happens to intercept images of either of these sizes, can immediately flag them as suspicious.

Of course, not every image of these sizes will have been altered, and it is very simple to modify a stego-system such that it can operate on images of any size. This means that a steganalyst will typically need to obtain a better range of features that represent suspicious images, often by digging deeper into how the stego-systems operate and evaluating the consequences of the embedding strategy. For example, after analysing the

Hide & Seek method thoroughly, a steganalyst may be drawn to the conclusion that a stegogramme has a larger than expected file size. Any image that is intercepted where this holds true is likely to be flagged as suspicious.

Structural attacks rarely analyse each image on its own merits. Instead, the images are scanned to see if they contain any of the known side-effects for various steganographic algorithms. Images that contain these properties are often subjected to further investigation. There are sometimes cases where the image may possess symptoms of steganography when it is actually perfectly innocent. For example, computer generated images are likely to have a different colour composition than those of natural life because they are not influenced by the same elements such as light, shadowing, and sampling[4]. Computer generated images may therefore appear structurally similar to what is expected for a stegogramme, but they do not necessarily contain hidden messages; this is why a more thorough investigation usually follows a structural attack.

4.2.1 Structural Attacks on File Size

Some image formats work differently from others in terms of how the pixel data represents a colour. As we have seen, grayscale images allocate 1 byte (8 bits) for each pixel so the value will fall in the range 0-255, where 0 represents a black pixel and 255 represents a white pixel. *Truecolour* images allocate 3 bytes per pixel (24 bits) where each byte represents the colour values (still in the range 0-255) for red, green, and blue respectively. If we take as an example, a truecolour image that is 1024x768 pixels in dimension (a typical size for high resolution images), we then have to multiply this by 3 to obtain the values for each colour plane. This means that the complete image is expressed via 2,359,296 bytes of information, which is over 2Mb. Truecolour images are very desirable for steganography as they contain an excess of redundant data that can be tweaked to embed a hidden message, and steganalysts often flag these types of images as suspicious as a result.

4.2.2 Structural Attacks on Palette-based Steganography

To avoid structural attacks on file size, steganographer's introduced algorithms that operated on different image formats such as GIF and PNG. For these image formats, each pixel value acts as an index to one of several colours in a pre-determined *palette*. For GIF images, each pixel is a single byte of information meaning that there are 256 possible colours for the image. We can calculate the number of colours available by saying that an i-bit image gives 2^i colours in the palette [24]. The plan for steganography was to perform an LSB style embedding approach in a similar fashion to the *Hide & Seek* method. However, as we have seen, this strategy either increments the entire pixel value by 1 or keeps it the same, which produces a problem for palette-based images. This is because the palette is not ordered in any particular manner, so indexing value 114 might produce a sky blue colour, where as the value 115 might produce a deep red colour.

[4]Camera's will typically pad the data to produce extra detail in order to create 24-bit images [24].

S-Tools tried to avoid this issue by reducing the palette size to represent the most important 32 colours, and then pad the palette to 256 colours by creating colours that are near to the 32 most important colours. The theory was that if the LSB's are flipped when a palette exists such as this, there will be minimal distortion to the image as the colours are likely to be flipped to something very similar to their original colour. However, this concept introduced a big flaw in that steganalysts were able to look at the structure of the pallete, and if it is comprised of a large sample of similar colours, then this indicates that the image has been manipulated.

Romana Machado introduced a much better system called *EzStego* that reordered the original palette such that similar colours are placed next to each other before embedding [24]. The embedding process works line-by-line on the image, and embeds the LSB of the *sorted index* within the pixel values according to the message bit stream. As Figure 4.6 shows, this is equivalent to embedding the message bit stream directly, but it ensures that the colours are only ever changed to a near neighbour or not at all, thus minimising distortion. The palette is then flipped back when the embedding process is completed.

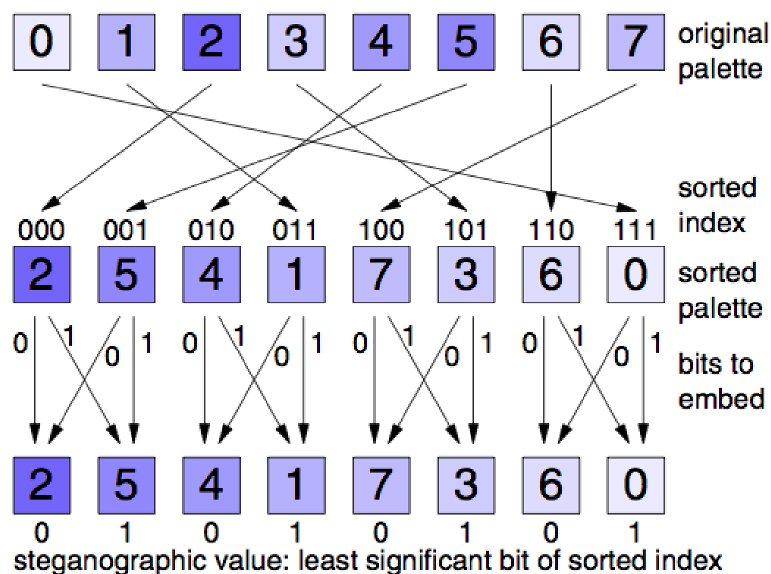


Figure 4.6: The embedding process of *EzStego* [26].

As we can see from Figure 4.6, if a pixel contains an index value of 2, and we want to embed a 1, the new index for that pixel becomes 5. If we were embedding a 0 however, then the value would not change.

What must be emphasised is that the colours expressed by the values 2 and 5 are very close together. Without sorting the palette first, but still embedding in the same fashion, the value 3 would have been obtained when embedding a 1. This colour is much lighter than value 5, so it would have been more obvious that some modifications had taken place. It should also be easy to see that when dealing with a much larger palette, this embedding technique has a stronger chance of success as the neighbouring colours in the

sorted palette will be even more similar to each other.

This approach has the obvious advantage over *S-Tools* in that no artificial colours are added to the palette. This means that if a steganalyst were to inspect the palette of the image, they would see nothing out of the ordinary and would be led to believe that the image is clean.

4.2.3 Summary

Steganography undeniably alters an image in such a way that will leave some difference between the cover Work and the stegogramme. Discovering what the difference is can be an excruciatingly difficult exercise, and - depending on the design of the stego-system - can often prove impossible. A successful structural attack relies on being able to identify a distinct difference between the clean image and a stegogramme, which means that there is a heavy reliance on either knowing the cover Work or knowing the intricate details of the embedding algorithm. It is rarely the case that a steganalyst will have access to one of these, and even rarer for them to have access to both, which only hampers the success of the attack.

As with visual attacks however, a structural attack can prove successful if the stego-system was so poorly constructed that it left gaping clues of manipulation. As discussed in section 4.2.2, *S-Tools* created a palette that could only be associated with bit-tweaking, and this meant that steganalysts could categorise images as stegogrammes if the palette took this form.

Structural attacks are arguably more important to steganalysts than visual attacks because they can be applied against a wider range of embedding techniques. The attacks work best when the steganalyst has access to a known stegogramme. In these instances, the steganalyst can probe the image for inconsistencies, thus producing a feature set that is known to be assoiciated with stegogrammes. A structural attack can then be instantiated by inspecting suspicious images for these features; those that contain the same properties are likely to be stegogrammes. With this in mind, structural attacks are rarely used as a means of proving that an image contains steganography, rather they highlight images that contain signs of embedding. As mentioned in the introduction to section 4.2, some images will carry the properties of stegogrammes without actually being one, so other methods of attack are often used to dig deeper into the properties of the image. This allows the steganalyst to conclude with a higher level of confidence whether or not the image has been tampered with.

4.3 Statistical Attacks

In mathematics, the study of *statistics* makes it possible to determine whether some phenomenon occurs at random within a data set [24]. Usually, a theory would be constructed that seemingly explains why the phenomenon occurs, and statistical methods can then be used to prove this theory to be either true or false. If we think about the data structure for a stegogramme, we can begin to see how statistics can be useful for steganalysis when proving whether or not the image contains a hidden message.

A stegogramme can be broken down into two data sets: *image data*, and *message data*. The image data relates to the information regarding the physical image that we can see, and will typically relate to pixel values that point towards the colours used in that region of the image. The message data on the other hand, relates to information regarding the secret message, and - if encrypted - it is typically more randomly composed than image data. It can safely be derived that the message data is more random than image data, and this is where statistical attacks usually operate. Whilst there is usually far less message data than image data, the small percentage of randomness created by the message data is enough to invoke an attack.

There are several methods that are known to prove the existence of a hidden message via statistical approaches; each aimed at identifying signs of embedding for specific stego-systems. In this section we will discuss some of the most effective statistical attacks, as well as providing details of why they are effective at defeating the stego-systems.

4.3.1 The Chi-squared (x^2) Test

One of the simplest and most popular statistical attacks is the *Chi-squared Test* (also referred to as the x^2 *Test)* which was documented in steganalytical terms by Andreas Westfeld and Andreas Pfitzmann in 1999 [26]. The test makes it possible to compare the statistical properties of a suspect image with the theoretically expected statistical properties of its carrier counterpart such that it is possible to determine the likelihood that a suspect image is a stegogramme.

Perhaps one of the reasons that the x^2 test is so widely respected is the fact that it is not necessary for the steganalyst to have access to the cover Work in order for the test to function. As mentioned earlier, the steganalyst will rarely have the luxury of obtaining the original cover Work, and so the main focus of the x^2 test is to be able to construct a method for accurately calculating the expected statistical properties of the original cover Work, without actually accessing it. To do this successfully usually requires a deep knowledge of various embedding strategies; here lies the reason why the test is categorised as a *targeted* approach. If the steganalyst is aware of a possible steganographic embedding strategy, then they are able to analyse the implications of embedding such that they eventually derive a set of features that can be analysed to determine the likelihood that a suspect image is a stegogramme.

To illustrate this by example, let us consider the x^2 test for sequential LSB steganography[5]. When we overwrite the least significant bits of an image with data from a message bit stream, we transform the values into each other. Let us consider an image where the first pixel value = 166. If we embed a 0, the value will remain the same, where as if we embed a 1, the pixel value will change to 167. Now let us assume that a pixel value of 167 exists somewhere else within the image. If we embed a 0 now, the value will be decreased to 166, where as embedding a 1 will leave the pixel value unchanged. What we witness here is that the two pixel values 166 and 167 can effectively be changed into

[5]A heavily documented x^2 test against sequential LSB steganography is known as the *POV3* algorithm which was designed in accordance with the PoV characteristic that occurs as a result of bit-flipping. The methodology of this algorithm is referred to in detail in [23].

each other, and the observation is true for all odd and even values in an image. We can therefore declare each odd and even value to be pairs, known in steganalysis as *Pairs of Values* (PoVs), where the PoVs for an 8-bit grayscale image would comprise of pixel values $\{0 \Leftrightarrow 1, 2 \Leftrightarrow 3, ..., 252 \Leftrightarrow 253, 254 \Leftrightarrow 255\}$. Flipping LSBs in this fashion will affect the frequencies of each pixel value (Y_k), while the sum of $(Y_k + Y_{k+1})$ will not be altered [23]. We can therefore calculate the expected distribution of the sum of neighbouring values for a clean version of the image to be:

$$E(e_i) = \frac{1}{2}(Y_{2i} + Y_{2i+1}) \tag{4.3}$$

Now that we have effectively obtained the expected distribution for the cover Work from equation (4.3), we can compare the result against that of the suspect image by using the x^2 formulae in equation (4.4) with $v - 1$ degrees of freedom:

$$x^2 = \sum_{i=1}^{v} \frac{(e_i - E(e_i))^2}{E(e_i)} \tag{4.4}$$

Substituting equation (4.3) into equation (4.4) directly, produces:

$$x^2_{PoV} = \sum_{i=1}^{127} \frac{Y_{2i} - \frac{1}{2}(Y_{2i} + Y_{2i+1})^2}{\frac{1}{2}(Y_{2i} + Y_{2i+1})} \tag{4.5}$$

The method can even be taken a step further by calculating the probability that an image is a stegogramme; the result is known as the *p-value*. It is calculated by scanning the image in the same order as the embedding algorithm, and by using a density function with x^2_{PoV} as its upper limit such as that illustrated in equation (4.6).

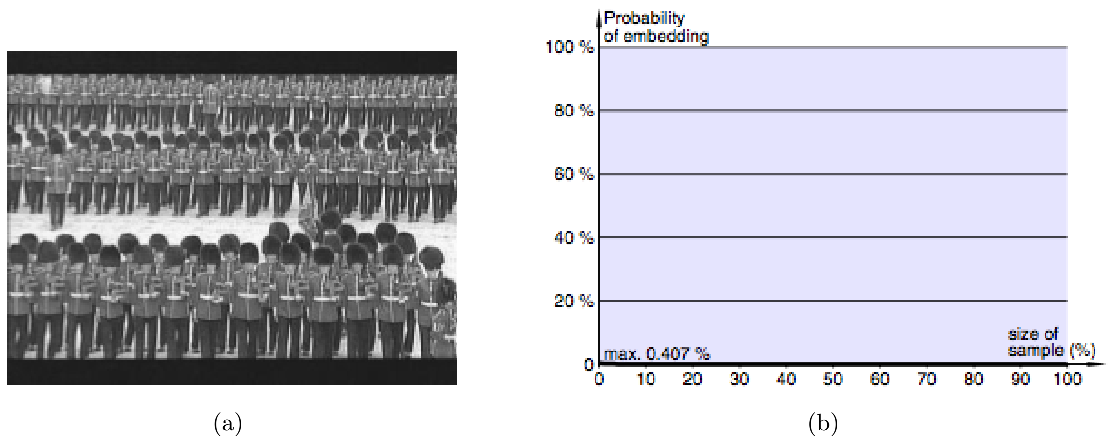
$$p(x^2_{k-1} \geq x^2_{PoV}) = 1 - \frac{1}{2^{\frac{k-1}{2}} \Gamma\left(\frac{k-1}{2}\right)} \int_0^{x^2_{PoV}} e^{-\frac{x}{2}} x^{\frac{k-1}{2}-1} dx, \tag{4.6}$$

where k = PoV, and Γ is the Euler Gamma function [23].

According to Westfeld [26], a stegogramme that contains $<$ 100% sequential embedding will yield p-values that are $\approx$1 when the manipulated regions are reviewed, and will dramatically descend to 0 when the normal image data is reviewed. This means that a x^2 test not only makes it possible to determine the probability that an image is a stegogramme, it is also possible to estimate the respective message length.

Figure 4.7 illustrates the p-value plot for a clean JPEG image. Note how the probability of embedding is constantly fixed to $\approx$0% (the peak value being 0.407%). This is because none of the desired properties were found when scanning through the image. If the steganalyst is presented with a plot similar to that of Figure 4.7 then they should assume

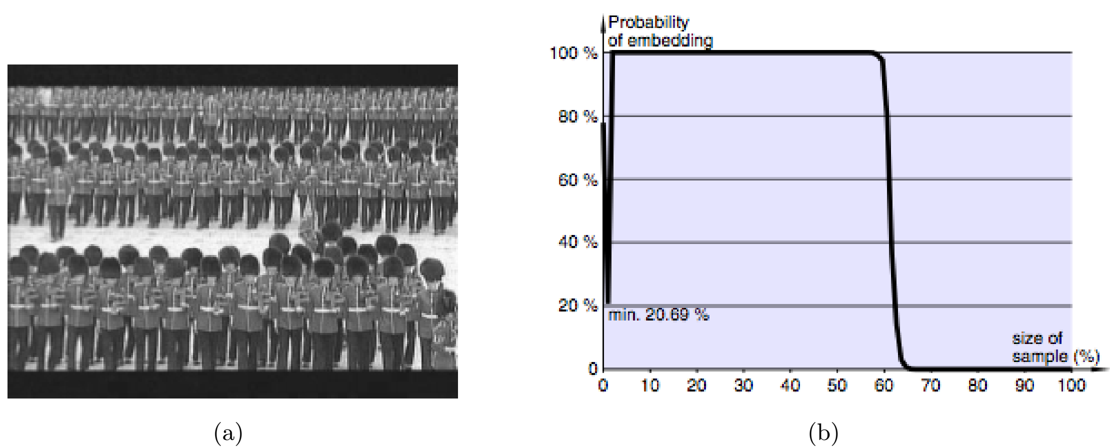
that the image has not been manipulated, at least not in the way that they are testing against.



(a) (b)

Figure 4.7: The probability of embedding for a clean image: (a) a clean JPEG image (0% embedding), (b) the corresponding p-value plot [26].

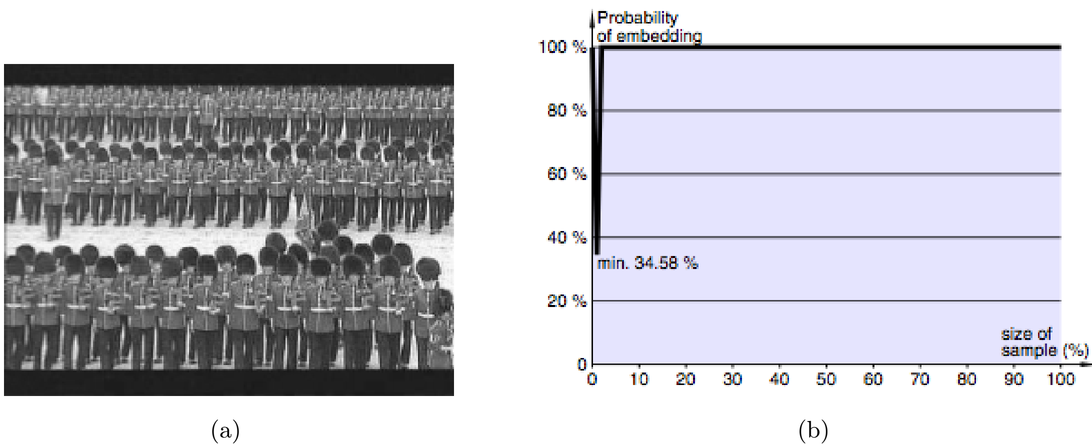
Figure 4.8 on the other hand illustrates the p-value plot for a stegogramme containing a hidden message at 50% embedding capacity. Here we can see that the probability of embedding is ≈100% whilst a hidden message exists. When the embedding ends, the p-value descends quickly to ≈0%, which means we can quite clearly predict the length of the message.



(a) (b)

Figure 4.8: The probability of embedding for a stegogramme created using *JSteg* [19] at 50% embedding capacity: (a) the stegogramme, (b) the corresponding p-value plot [26].

Finally - for completion - let us consider the stegogramme in Figure 4.13 that contains a hidden message of 100% capacity. In direct contrast to the plot in Figure 4.7, we can see that the probability of embedding remains at ≈100% for the entire duration of the image. This is because the examined properties are matched all over the image, meaning the steganalyst can conclude the suspect image contains clear statistical signs of embedding. Also, the fact that the p-value does not deviate from ≈100% means that the steganalyst is also able to derive the message length to be at 100% capacity.



(a) (b)

Figure 4.9: The probability of embedding for a stegogramme created using *JSteg* [19] at 100% embedding capacity: (a) the stegogramme, (b) the corresponding p-value plot [26].

Attack methods such as these are adaptable to all forms of steganography that embed sequentially. Additionally, the method does not have to be applied in the spatial domain. If LSB embedding has taken place sequentially in the Discrete Cosine Transform for example, then the x^2 attack can be used to identify discrepancies in images in this domain. Steganographic algorithms such as *JSteg* embed a secret message in precisely this manner. To illustrate another example, the *EzStego* algorithm implements sequential LSB embedding within the colour palette of GIF images, meaning the x^2 test can be applied to the palette in order to determine the probability that a suspect image contains steganography.

4.3.2 The *Generalised* Chi-squared (x^2) Test

The x^2 test described in section 4.3.1 illustrates how steganalysts can successfully detect stegogrammes produced by sequential embedding approaches. The success of the test heavily relies on knowing the original order that the message was embedded in such that we can re-scan the values in the same order to yield the x^2 statistic. Several steganographic algorithms have been developed such that they randomise the embedding

approach (these algorithms include *OutGuess 0.1*, *OutGuess 0.2*, *F3*, *F4*, *F5*, etc.). By randomising the embedding sequence, it is impossible for the x^2 test to succeed, so several attempts have been made to *generalise* the approach such that it can still operate.

The most notable work in this area is the work by Provos and Honeyman [17] in 2001. Their approach modified the standard x^2 test by using a constant sample size but shifting the position where the samples are taken [28]. This approach is in contrast to the standard approach that increases the sample size and applies the test at a constant position. Whilst this *generalised* technique makes it possible to detect randomly dispersed data hiding, it renders the p-value calculation obsolete as there is no clear distinction between the embedded data, and standard image data. Instead, the p-value plot will fluctuate somewhat arbitrarily between 5% and 95%. This means that the generalised x^2 test is not capable of estimating the message length.

4.3.3 Histogram Attacks

As Andreas Westfeld discussed in [28], the *JSteg* algorithm introduces *Pairs of Values* (PoVs) as a result of sequential bit-flipping. It is possible to illustrate these PoVs by extracting all of an image's DCT AC coefficients and tallying their frequencies of occurrence. If we split the values into *bins* we can narrow the results to a focussed subsection and display the results by centering them across a specified range x. The result is referred to as a *histogram*.

What we expect to see for a clean image is that the histogram illustrates a linear distribution to the frequencies of the DCT coefficients across zero. As the values have not been altered by any embedding process, there is a clear structure to the values that provides a characteristic for detecting steganography.

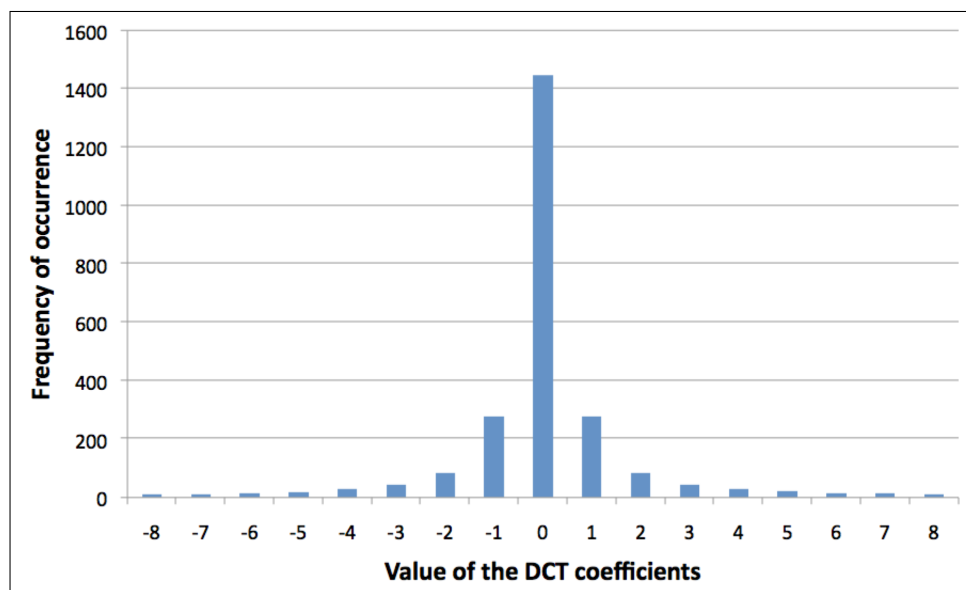


Figure 4.10: The histogram of a clean 8-bit JPEG image.

Figure 4.10 shows the histogram of a clean 8-bit JPEG image. As expected, the values increase in frequency in a linear fashion towards zero, and decrease after. If we compare this histogram with that of a *JSteg* stegogramme at 80% embedding capacity, we can see how important a role the PoVs play in steganalysis.

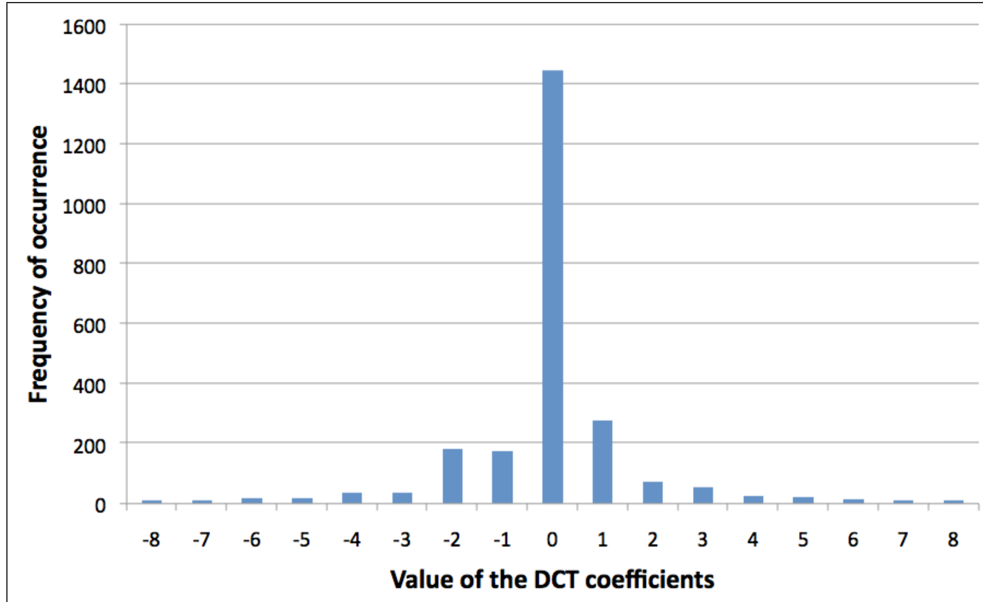


Figure 4.11: The histogram of an 8-bit stegogramme produced using *JSteg*.

Figure 4.11 shows that the PoVs created by *JSteg's* bit-flipping methodology are apparent in the stegogrammes histogram. All of the values (except 0 and 1 which *JSteg* does not embed within) can be paired together by their neighbouring values because their frequency of occurrence has become very similar. For example, the value -2 occurs roughly as often as the value -1, and similarly the value 2 occurs roughly as often as the value 3. This trait is characteristic of a stegogramme created by bit-flipping, and we would therefore expect to see a similar plot for stegogrammes created using *OutGuess 0.1*, and *F3*.

As we mentioned in Section 3.3.8 of Chapter 3, the *F3* algorithm effectively embeds more zeros than ones. When we view the histogram of resulting stegogrammes, we can see a much higher than expected frequency distribution for coefficients equal to zero. Figure 4.12 shows the histogram of a stegogramme produced using the *F3* algorithm (the same 8-bit cover image was used). As expected, there is a significantly large proportion of zeros in comparison with all of the other coefficients.

Also, as a result of *shrinkage*, the frequencies of even coefficients outweighs the frequencies of their neighbouring odd values (excluding zero and 1). This phenomenon develops because of the large number of zero (even) coefficients. In addition, the *F3* algorithm interprets coefficients equal to 1 or -1 as 1 because their LSBs are 1. This means that the frequencies of these values remain unchanged.

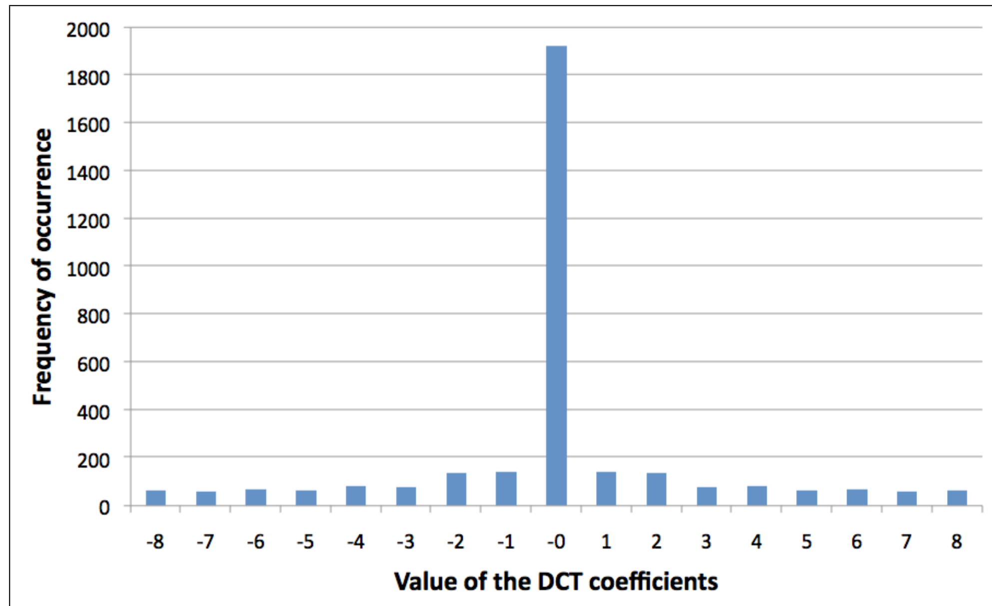


Figure 4.12: The histogram of an 8-bit stegogramme produced using *F3*.

Observation

When testing this statistical approach, it was apparent that the success of the attack depends not only on the PoVs created as a result of bit-flipping, but also on the *Quality Factor* (QF) that was used to compress the JPEG image.

257	6	3	0	0	0	-6	7
8	0	1	-5	2	4	-4	3
-5	-1	-1	1	-1	-1	2	-2
2	2	2	2	-1	-2	0	0
-1	-2	0	-2	2	1	-1	0
2	1	-2	0	-2	1	2	1
-2	0	3	2	2	-3	-1	-1
2	0	-2	-2	-1	2	0	1

(a)

86	3	1	0	0	0	-1	1
4	0	0	-1	0	0	0	0
-2	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b)

Figure 4.13: The affect of Quality Factors on the DCT domain: (a) 8x8 sub-block for an image where $QF = 90$, (b) 8x8 sub-block for an image where $QF = 80$,.

When experimenting with different QF values, a pattern emerged that showed that when using a QF < 85, more zeros are introduced as AC coefficients in each 8x8 DCT sub-block. In addition to this, there are few AC values $\neq 0$. Whilst this means that the length of the hidden message is heavily restricted (because some algorithms will not embed on 0s or 1s), it also means that less PoVs are created, making this attack less likely to succeed. This is in direct contrast to using a QF > 85, where the frequency of AC coefficients equal to zero, are reduced.

Figure 4.13 illustrates the above findings by displaying the 8x8 sub-blocks for two images that were both formed from the same source image, but using different Quality Factors. As we can see in image (a) (QF $= 90$), relatively few AC coefficients equal 0. However, image (b) shows that when the QF is reduced to 80, more zeros are formed, whilst the frequency of AC coefficients $\neq 0$ is largely reduced. If we consider that a Histogram Attack relies on a pattern being formed when embedding in non-zero AC coefficients, the pattern will be negatively affected by low QFs that produce a magnitude of zero values.

4.3.4 Pairs Analysis

The statistical attacks mentioned earlier have each focused on JPEG-based steganography. In order to show that statistical attacks can work on a wide range of embedding techniques, we should discuss briefly *Pairs Analysis* which was designed for use against palette-based stego-systems such as *EzStego* (although, as Andrew Ker suggests in [12], there is no good reason why the technique cannot be extended for use with grayscale images as well).

The approach is to form *colour cuts* by scanning an image and selecting only the two colours c and c', assigning 0 to c and 1 to c' in order to build up a binary vector $Z(c, c')$. This can now be referred to as the colour cut for the pair c and c'. As palette images have relatively few colours, there will be a distinct structure for Z that will change if embedding has taken place; according to research in [7], the embedding procedure will increase the entropy of Z. Colour cuts are obtained for all colour cuts before the results are concatenated to a single vector $Z = (0,1) + (2,3) + ... + (252,253) + (254,255)$. When values are changed into each other as a result of bit-flipping, the impact is that the structure of the colour values is damaged. As a result, we also calculate the shifted pairs $Z' = (1,2) + (3,4) + ... + (253,254) + (255,0)$ [21].

The *homogenous pairs* (00) and (11) are counted for the vectors so that we can determine the expected relative count of homogenous pairs $R(q)$ in Z after flipping $q(\%)$ values. According to [7], $R(q)$ is a parobola with its vertex at 0.5 and R($\frac{1}{2}$). As this is the case, we can calculate $R(q)$ by dividing the count of homogenous pairs in Z with the total pixel count for the image.

The method makes it possible to determine the embedding strength based on the assumptions of the PoV structures for a clean image, and comparing them against the results of the above attack. This has the main advantage that the clean image is never physically investigated, instead it is derived from a pattern developed by the bit-flipping embedding strategy.

4.3.5 Summary

We have briefly seen how statistical attacks operate against specific statistical traits of images. Some of the attacks are easier to implement than others, but the hardest part is understanding the theory behind them.

Statistical attacks are often preferred to visual attacks and structural attacks because they can be automated. This means that there is less pressure on the steganalyst to determine whether an image is a stegogramme or not because the computed result essentially does this on its own. Computed decisions should greatly reduce the total frequency of false-negative results as they are not prone to personal interpretation as visual attacks are. The results are usually so clear-cut that they can only be interpreted in one way.

Another benefit of statistical attacks is that they do not require a deep knowledge of what the cover image should look like. Whilst for structural attacks this was a very big part of the success, statistical attacks simply form an analysis based on what is presented from the suspect image alone. However, a deep knowledge of various embedding algorithms is important. If the steganalyst knows a wide selection of embedding tools then they can better design a statistical attack that identifies the artifacts of their embedding process.

As a result of the success of statistical attacks, stego-systems were developed that preserve the statistics of the clean image. Such stego-systems are often referred to as *statistics-preserving.* The best example of these systems is the *F5* algorithm which looks to rebuild the structure of an image such that the histogram appears normal. *OutGuess 0.2* operates in much the same way, making the likelihood of successful detection much less likely.

4.4 Conclusion

The steganalytical methods discussed in this chapter have provided an overview of a number of *targeted* schemes, designed in direct accordance with a specific method of embedding. Some of the attacks are more successful in others, and we have deliberately listed the attacks in order of usefulness such that the reasons become apparent.

The first type of attack we reviewed was Visual Attacks. We found that whilst attacks of this nature are extremely trivial to implement, classifying the suspect image as either a stegogramme or an innocent image is less so. The success of the attack not only depends upon knowing the exact regions of the suspect image that have been manipulated to inhibit message data, it also depends upon a *sequential* embedding strategy such as *Sequential Hide & Seek*, and *JSteg.* If the message data is distributed randomly over the stegogramme, then the steganalyst has a dramatically reduced chance of identifying the artifacts of embedding, particularly when the cover image is not available. Of course, most algorithms can easily be adapted such that they eliminate at least one of these factors. We have seen how easy it is to shuffle the pixel locations before embedding, thus effectively embedding in random locations. We also saw in Chapter 3 that the more advanced stego-systems embed the message data in more discrete areas of the image, which will make it harder for the steganalyst to identify which regions of the image to

examine. Whilst Visual Attacks were successful for early steganalysis, the complexity of the stego-system encoders has evolved such that the attack is now too basic to be useful.

We also reviewed Structural Attacks which are arguably more useful for steganalysts, although only just. A Structural Attack makes it possible to pin-point a data-level feature that changes as a result of embedding message data. The most obvious example is the file size of an image that will typically increase or decrease depending on the embedding algorithm. The steganalyst can then flag any image that contains a higher or lower file size than expected as a *suspicious* image, where further analysis is usually performed. Structural Attacks are more useful than Visual Attacks because they are more difficult to contain from a steganographic point of view. This means that there are potentially more stego-systems that exist where a structural attack can be applied successfully, although the more recent stego-systems will still be too advanced for the attack to work.

The final type of attack we reviewed was Statistical Attacks. These are highly regarded as the most successful form of targeted steganalysis because they are capable of making an automatic classification as to the integrity of the image, based on what we expect to see for a clean image, and what is actually seen from the suspect image.

5 Blind Steganalysis

As we discussed in Chapter 4, there are several methods available for attacking a stegogramme via a *targetted* approach. In order for these attacks to work effectively however, the steganalyst ideally needs to know how the stegogramme was created (referred to as a *known-stego* attack) such that they can apply the correct attack. Either this, or they will need to have access to the original cover image (referred to as a *known-cover* attack) in order to calculate the expected properties for a clean image, and make a comparison with what is seen for the stegogramme.

As time goes on, more and more stego-systems are created, and we cannot be sure how all of them operate. Subsequently, an approach is needed that is capable of identifying the probability of embedding, even when we are not sure how the information might have been embedded. This is known as *blind* steganalysis.

Blind steganalysis therefore works differently to *targeted* steganalysis because it assumes that nothing is known about either the algorithm or the cover image that was used to produce a suspect image. The attacks attempt to evaluate the probability of embedding based solely on the data of the suspect image. Such approaches are more common in real-world steganalysis because a steganalyst will rarely know much more about an image than what they can extract from the image itself.

5.1 Early Methods for Blind Steganalysis

One of the earliest *blind* steganalysis techniques was developed by Nasir Memon [15] who used Image Quality Measures (IQM) to train the system such that it could classify images according to the likelihood that they contain message data. The approach made way for a large amount of research in Neural Network steganalysis.

Hany Farid [4] also suggested a method based on features extracted from the wavelet (transform) decomposition of the suspect image. His conclusion stated that there are strong high-order statistical regularities within these regions for natural images, and that these regularities are altered as a message is embedded.

Jessica Fridrich et al. [5] provided an easier method for *blind* steganalysis based on *self-calibration*. This chapter describes this process, explaining how it is possible to produce an estimate of the cover image using only the suspect image. By using this cover estimate, the steganalyst can perform more generalised attacks than those described in Chapter 4, and effectively calculate the probability that the suspect image is a stegogramme.

5.2 JPEG Calibration

5.2.1 Overview

Perhaps the most important aspect of *blind steganalysis* is ensuring that we can derive an estimate of the cover image that is as accurate as possible. The attacks that follow this procedure often compare the data in the estimated cover image to that of the suspect image, so it is imperative that the data of the estimate is as sound as possible so as to not obscure the results.

One of the most famous approaches for creating an estimate of the cover image is the model proposed by Jessica Fridrich in [5] known as *JPEG Calibration*. The method takes advantage of the fact that most stego-systems encode the message data in the transform domain during the compression procedure to produce JPEG stegogrammes. Given that the JPEG compression algorithm operates by transforming the image into 8x8 blocks, and it is within these blocks that the encoding of the message operates, we can estimate the cover work by introducing a new block structure and comparing it with that of the suspect image. When there is a large difference, it suggests that the suspect image is a stegogramme, where as little difference typically indicates that the image is innocent.

5.2.2 Methodology

The general methodology of the *calibration* process decompresses the suspect image using its quantisation table, removes 4 pixels from each side, and then recompresses the result using the same quantisation table. Visually, and technically (by measures such as PSNR[6]), the calibrated image is still very close to that of the suspect image. However, as a result of cropping the image and recompressing, we effectively break the block structure of the suspect image because the second compression does not consider the first. Figure 5.1 shows a graphical representation of the embedding process.

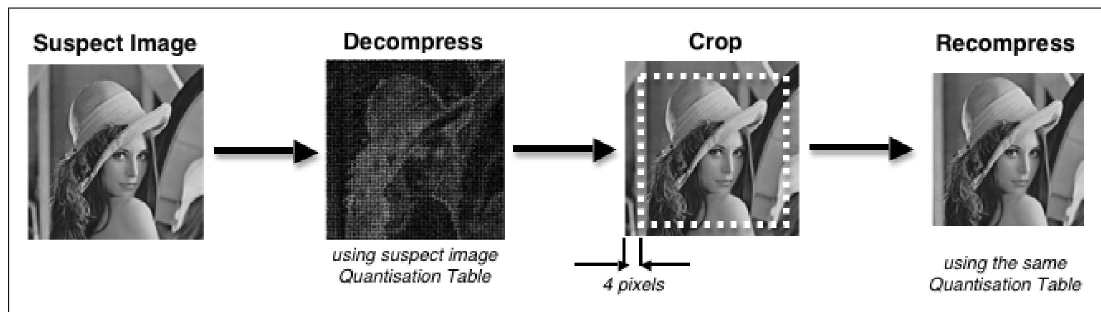


Figure 5.1: The calibration process.

The net result is that the image will inherit a more genuine block structure as the compression was not influenced by the message data included in the quanitised DCT

[6]PSNR (Peak Signal to Noise Ratio) is used to measure the difference between one image and another

coefficients. The block structure will therefore look more like what we would expect to see for an innocent image.

5.2.3 Comparing The Calibrated Image Against The Original Image

The purpose of calibration is to produce an accurate estimate of the original cover work, such that we can run an attack as though the calibrated image is actually the cover image. Therefore, the calibrated image should be extremely close to the cover image in at least one statistical aspect in order to successfully determine the probability of embedding for the suspect image.

Perhaps the most effective way of determining how similar the images are is to compare the histograms of them both and overlay the plots such that we can see how similar they are. Figure 5.2 shows the histograms of the cover image, the calibrated image, and also the histogram of the stegogramme. The stegogramme was created by embedding a message at 50% capacity using the *F4* algorithm.

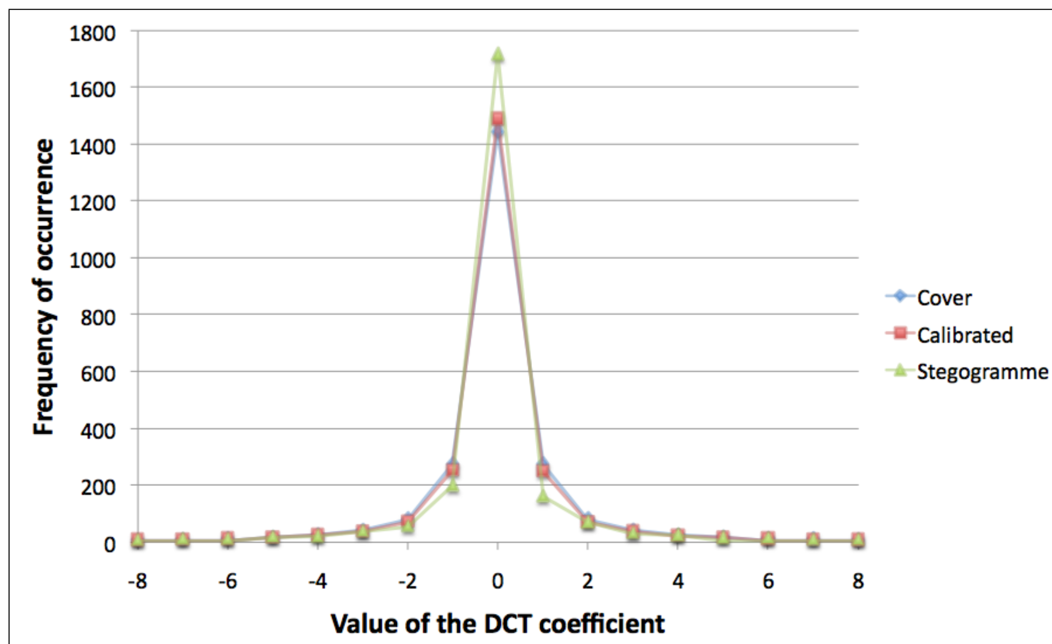


Figure 5.2: Comparing the histogram of the calibrated image with that of the cover image.

As we can see from Figure 5.2, the histograms of the cover image and the calibrated images are very close together, meaning the calibration process has been successful. We have managed to create an image that contains roughly the same statistical property of the original cover image, even though we had no access to the cover image at any point. We can also compare these two histograms against that of the stegogramme. Note that the histogram for the stegogramme is much more distant from the cover image and calibrated image. If we were to eliminate the plot for the cover image in Figure 5.2, we

would be left with two rather varying plots. We could make a guess that the suspect image is a stegogramme based on this information, but as the calibration process relies heavily on the information of the quantised DCT coefficients, we should not use this histogram alone to make a final decision. However, we have illustrated a blind steganalytical method for evaluating the probability that the image is a stegogramme.

Observation

When testing the calibration process, it became apparent that the best methodology was to crop the image by 4 pixels in every direction (top, bottom, left, and right). Some literature suggests that 4 pixels should be cropped from the left-hand side of the suspect image, and a further 4 pixels should be cropped from the right-hand side of the image. However, this method does not remove the block structure as well as it should, as this is only equivalent to a half block shift to the side; the block structure from top to bottom remains intact. Cropping from all edges ensures that the entire block structure is removed, and thus a more accurate estimation is derived.

5.3 Blockiness

Now that we can derive an estimate of the cover image, we need to find some statistical property that differs between the calibrated image and the suspect image such that we can determine the probability that the image is a stegogramme.

One of the strongest methods for achieving this is known as *Blockiness* which takes advantage of the fact that JPEG-driven stego-systems encode the message data in the same 8x8 blocks that are used for compression. The method is defined best by Dongdong Fu in [8] when it is stated that:

"*[Blockiness] defines the sum of spatial discontinuities along the boundary of all 8x8 blocks of JPEG images*".

Essentially, the logic behind Blockiness is that a stegogramme will contain a different set of coefficients across the boundaries of each 8x8 block to that of a clean image. We can therefore total the sums of the boundaries column-wise and row-wise for both a suspect image and a clean image (or our calibrated image) and then calculate the difference between the two. A large difference suggests that the image is a stegogramme, whilst a small difference is probably down to compression, and therefore reflects a clean image. The formula for calculating the Blockiness of an image is shown in equation (5.1).

$$B = \sum_{i=1}^{\left[\frac{M-1}{8}\right]} \sum_{j=1}^{N} |g_{8i,j} - g_{8i+1,j}| + \sum_{j=1}^{\left[\frac{N-1}{8}\right]} \sum_{i=1}^{M} |g_{i,8j} - g_{i,8j+1}| \tag{5.1}$$

where $g_{i,j}$ refers to the coordinates of a pixel value in an MxN grayscale image [20].

As we can see from equation (5.1), the formula operates in a column-wise and row-wise motion rather than calculating the blockiness for each 8x8 block individually. This is achieved by firstly calculating the sum of the values for the 8th row, and then calculating the sum for its neighbouring row (row 9). This process is then repeated for every row-wise multiple of 8, where the each sum is added to the accumulated total until the sums of all the rows have been calculated. The same method is then instantiated for the columns, before finally adding the two totals together. This value is the Blockiness of the image.

Since each 8x8 block positioned around the edges of an image do not have boundaries on every side, the blockiness formula does not calculate the sum of the boundaries for these blocks.

To express this in graphical terms, consider Figure 5.3. It firstly shows the boundaries of the 8x8 blocks in (a), and then shows what these values look like in the spatial domain in (b). The red lines indicate the columns that are multiples of 8, and the yellow lines represent their neighbouring columns that are multiples of $8+1$. For each column, the sum of the yellow column is subtracted from the red column. Similarly, the sum of the green rows are subtracted from the blue rows. The absolute values of the two separate totals are then added together to yield the blockiness value.

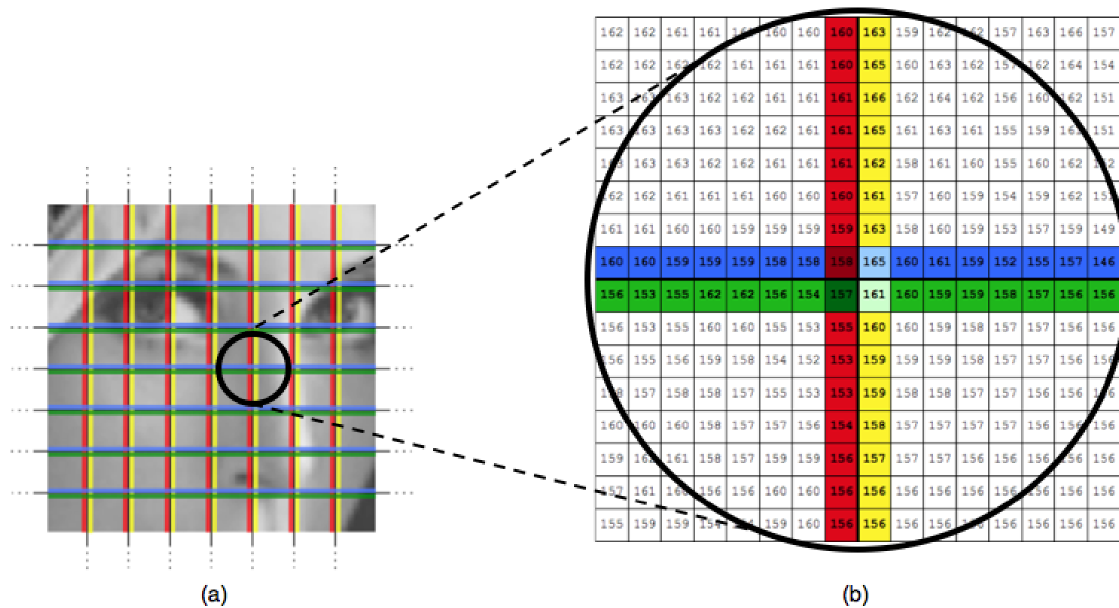


Figure 5.3: Graphical representation of the blockiness algorithm.

Algorithm 13 shows an example of how the Blockiness algorithm can be implemented using MATLAB. Again we can see that the absolute sum of the boundary differences for both the rows and columns are accumulated, before being added together at the end to form the Blockiness value.

Algorithm 13 The Blockiness algorithm in MATLAB (adapted from [14].

```
[M N] = size(image)                   % take height and width of image

horizontal = 0;                       % initialise row variable
vertical = 0;                         % initialise column variable

for i=1:(M-1)/8                       % for every 8th row
    upper = image(8*i,1:N);           % store all upper-boundary values
    lower = image(8*i+1,1:N);         % store all lower-boundary values
    total = sum(abs(upper-lower));    % absolute sum of the difference
    horizontal = horizontal+total;    % accumulate sums for each iteration
end

for j=1:(N-1)/8                       % for every 8th column
    left = image(1:M,8*j);            % store all left-boundary values
    right = image(1:M,8*j+1);         % store all right-boundary values
    total = sum(abs(left-right));     % absolute sum of the difference
    vertical = vertical+total;        % accumulate sums for each iteration
end

Blockiness =  horizontal+vertical;    % calculate Blockiness
```

5.4 Forming An Attack Through Calibration And Blockiness

5.4.1 Overview

As we mentioned in section 5.2, the calibration process is simply a means of estimating the cover image entirely from the suspect image. Subsequently, the steganalyst should always be aware that it is only an estimate of the cover image, and should never be assumed to be identical.

However, it is possible to perform steganalysis such that we can yield solid results from the calibrated image. This was achieved by Jessica Fridrich et al. in [5] when a process was described for calculating the Blockiness of both the calibrated image and the suspect image after further embedding messages of varying capacities, such that the *OutGuess 0.2* algorithm, and the *F5* algorithm could both be attacked. There was no previous approach known for achieving this.

This chapter will describe the logic and methodology that makes this attack possible. We will demonstrate each stage of the process by using a stegogramme produced by *OutGuess 0.2* from a grayscale 512x512 image, with a message load at 50% capacity. We will also create a calibrated version of this image, produced by following the calibration procedure outlined in section 5.2.

5.4.2 The Logic of the Attack

If we think about how the targeted steganalytical attacks operated in Chapter 4, we can infer that they all function by firstly uncovering a property of the cover image that is likely to change as a result of encoding a secret message. The statistical approaches then estimated the expected results for this property if the image were clean, and then compared them against the results obtained from the suspect image. If there was a large difference then the image was likely to be flagged as a stegogramme, otherwise it would likely be considered clean. The logic is very similar when the attack is applied to a suspect image and the calibrated image that is an estimate of the original cover image.

The attack attempts to find some *macroscopic quantity* $S(p)$ that changes predictably as the embedding capacity p changes for the image. For example, a good statistic for $S(p)$ can be derived when the yielded result increases monotonically as the embedding capacity p is increased. When this is the case, we are able to evaluate *extreme* values for S, such as $S(1)$ and $S(0)$ that represent the statistics derived from embedding our own message at full capacity, $S(1)$, and for the estimated cover image (with no embedded message), $S(0)$. Once these extreme values have been calculated, we can obtain the value for the suspect image. Depending on where the result falls in relation to $S(1)$ and $S(0)$ determines whether the suspect image should be classified as a clean image or a stegogramme. For Outguess 0.1, histogram statistics could be used as S, but Outguess 0.2 made this impossible by mimicking the histogram statistics of the original cover image after embedding. Thus a new macroscopic quantity needs to be used, and this is where the Blockiness comes in.

To put this attack in view for Blockiness, we need to calculate the extreme values based on the Blockiness results obtained for the suspect image $S_{suspect}$. If Blockiness B is our macroscopic quantity, then we can obtain $B_{suspect}(0)$ by calculating the Blockiness of the suspect image as it is received, and we can obtain $B_{suspect}(1)$ by further embedding our own message at 100% capacity and calculating the Blockiness value of the result. The statistic used for measuring the probability of embedding is referred to as the *distinguishing statistic* S, and in terms of the above methodology, it can be derived by solving equation (5.2).

$$S_{stego} = B_{suspect}(1) - B_{suspect}(0). \tag{5.2}$$

In order to obtain the strongest results in terms of how confident we are about the state of the suspect image, we should calculate Blockiness at different embedding capacities p. This will not only prove that B monotonically increases in relation to p, but it will also prove that the results for $B_{suspect}(0)$ and $B_{suspect}(1)$ are not erroneous. We therefore calculate the results for three separate images: S_0, S_q, and S_1. S_0 refers to our calibrated image, S_q refers to the suspect image (with an embedded message of an unknown length q), and S_1 refers to an image that we can create by embedding our own message on top of the suspect image, at 100% capacity. We are then essentially able to calculate B for a clean image, a fully embedded image, and the suspect image (which we will hope is somewhere between the two). By further embedding messages at varying embedding

capacities, we can plot the slopes for each image such that we are able to derive q from the suspect image. This will provide us with enough information to evaluate the probability of embedding for the suspect image.

5.4.3 A Working Example

To illustrate how the Blockiness and Calibration techniques can be used to attack *OutGuess 0.2*, we will discuss the step-by-step approach of obtaining values for the three images: S_0, S_q, and S_1. The suspect image is actually a stegogramme created using the *OutGuess 0.2* algorithm, by embedding a message of 50% capacity onto a 512x512 RGB image. This image will be referred to as a *suspect* image S_q in this section as this is all the steganalyst will typically treat the image as when it is intercepted. We then calibrate this image according to the process described in section 5.2 to produce S_0. Finally, we embed a message of 100% capacity on S_q using sequential *Hide & Seek* to produce S_q. We will then embed further messages of increments of 10% capacity and calculate the Blockiness of each resulting image in order to plot the slopes for each of our three image types.

We will first concentrate on obtaining the values for the calibrated image. If the Blockiness algorithm is a good measure, we should find that the results monotonically increase as the embedding capacity p increases. The table in Figure 5.4 shows the results obtained from this test. Please note that the Blockiness results have all been normalised because the calibrated image is smaller than the suspect image (because of cropping), and therefore has fewer 8x8 blocks to process. If we did not normalise the results, we would not be able to compare the calibrated image with the other images at the end of the analysis.

Message length (q%)	Blockiness (normalised)
0	204064
10	205475
20	206669
30	208572
40	209748
50	211185
60	211559
70	212957
80	214586
90	216972
100	218276

Figure 5.4: Blockiness results for calibrated image for increasing embedding loads.

By plotting these results, we obtain Figure 5.5.

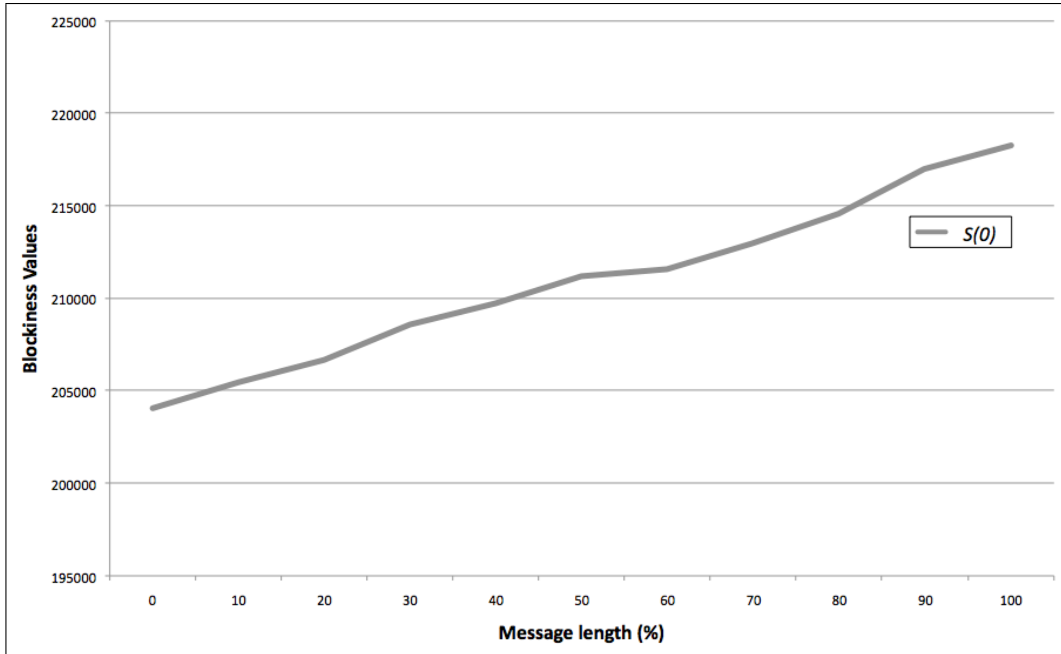


Figure 5.5: Plot of Blockiness values for $S(0)$ for increasing embedding loads.

As we can see from the results shown in Figure 5.4 and Figure 5.5, the Blockiness values increase as the message length is increased. This means that the Blockiness measure is giving us the desired results, and means that it is proving to be a useful macroscopic quantity.

The same calculations are now performed on the suspect image $S(0)$ by further embedding messages of varying capacities q and noting the Blockiness values for each resulting image. We again expect to note that the Blockiness values increase as q is increased.

Message length (q%)	Blockiness (normalised)
0	211859
10	212910
20	213461
30	214310
40	214575
50	215942
60	217573
70	217635
80	218118
90	219333
100	220111

Figure 5.6: Blockiness results for suspect image for increasing embedding loads.

By plotting these results, we obtain Figure 5.7.

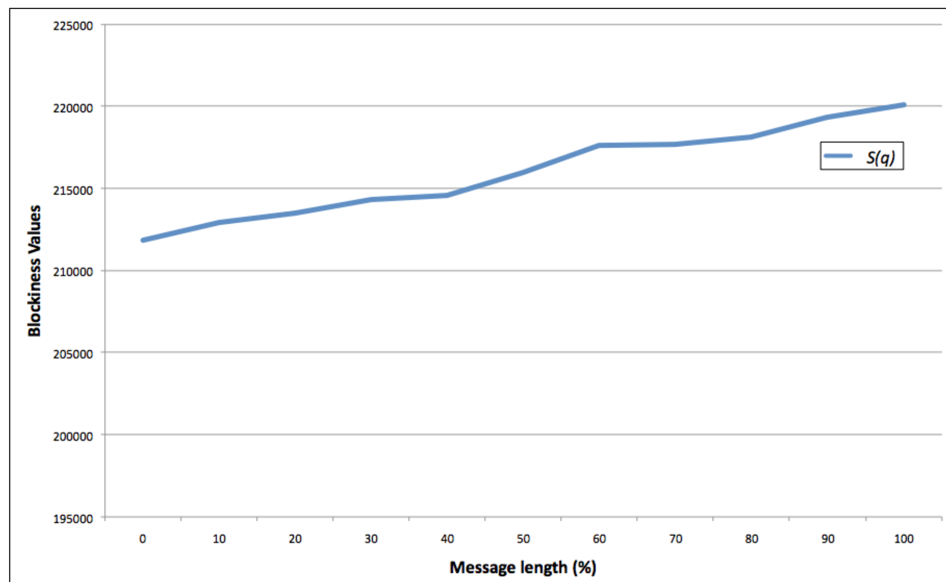


Figure 5.7: Plot of Blockiness values for $S(q)$ for increasing embedding loads.

Again, we can see from Figure 5.6 and Figure 5.7 that the Blockiness results for S_q increase as we increase the embedding load, as we also found for the results from $S(0)$. If we look closely at the two data sets for $S(0)$ and $S(q)$, we can also see that $S(q)_q > S(0)_q, \forall q$. This is a good sign that the process will be successful as it has essentially proved that the Blockiness of the suspect images is higher than that of the cover image, for every length q that we embed. This makes perfect sense from a steganalytical viewpoint as we expect that there are more discontinuities in each 8x8 block for any image that includes message data; the Blockiness results we are obtaining, seem to reflect this assumption.

Finally, we need to obtain the Blockiness results for our second extreme value, $S(1)$ which contains a message that we have embedded at 100% capacity. We then apply the same methodology as above, and yield the blockiness values for each image.

Figure 5.8 and Figure 5.9 both show that the distribution of the Blockiness values for $S(1)$ is more random than we have seen for $S(0)$ and $S(q)$. This is to be expected, as we are embedding on top of an image that already contains maximum embedding, which in turn yields high Blockiness values for every q. This can be accounted for by the fact that the *OutGuess 0.2* algorithm embeds the message data randomly, and we are embedding sequentially using the *Hide & Seek* algorithm. If the original message (embedded using *OutGuess 0.2*) were $<$ 100% capacity, then we will only be overwriting some of the message data some of the time. Thus, this causes only a small change on the Blockiness value for each q.

Message length ($q\%$)	Blockiness (normalised)
0	220111
10	220671
20	220644
30	220789
40	220649
50	220620
60	220690
70	221015
80	221583
90	221654
100	220801

Figure 5.8: Blockiness results for $S(1)$ for increasing embedding loads.

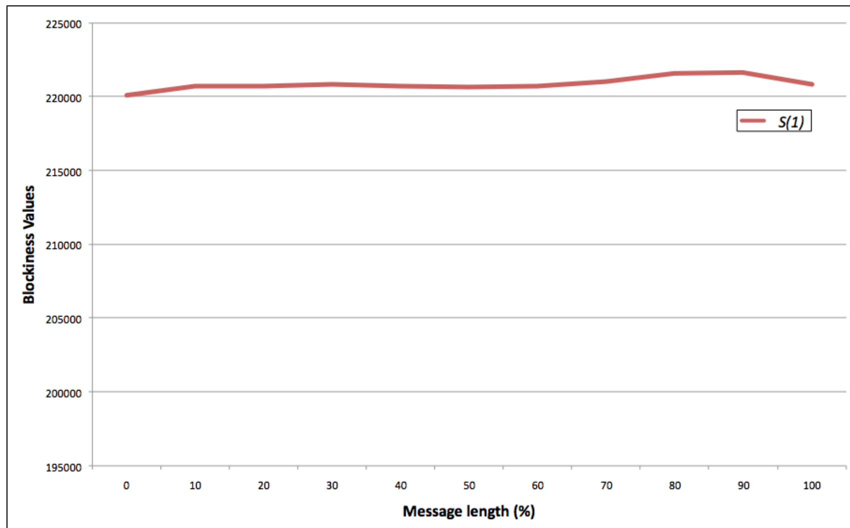


Figure 5.9: Plot of Blockiness values for $S(1)$ for increasing embedding loads.

What we can note however, is that $S(1)_q > S(q)_q, \forall q$. As we also found that $S(q)_q > S(0)_q, \forall q$, this means that our two extreme values, $S(0)$ and $S(1)$, sandwich the results of $S(q)$. Consequently, as $S(0)$ estimates the Blockiness values for a clean image, and $S(1)$ estimates the Blockiness values for a fully loaded stegogramme, we can infer that our suspect image $S(q)$ is a stegogramme because its Blockiness values constantly fall between $S(0)$ and $S(1)$. Figure 5.10 shows the plots for all three images together.

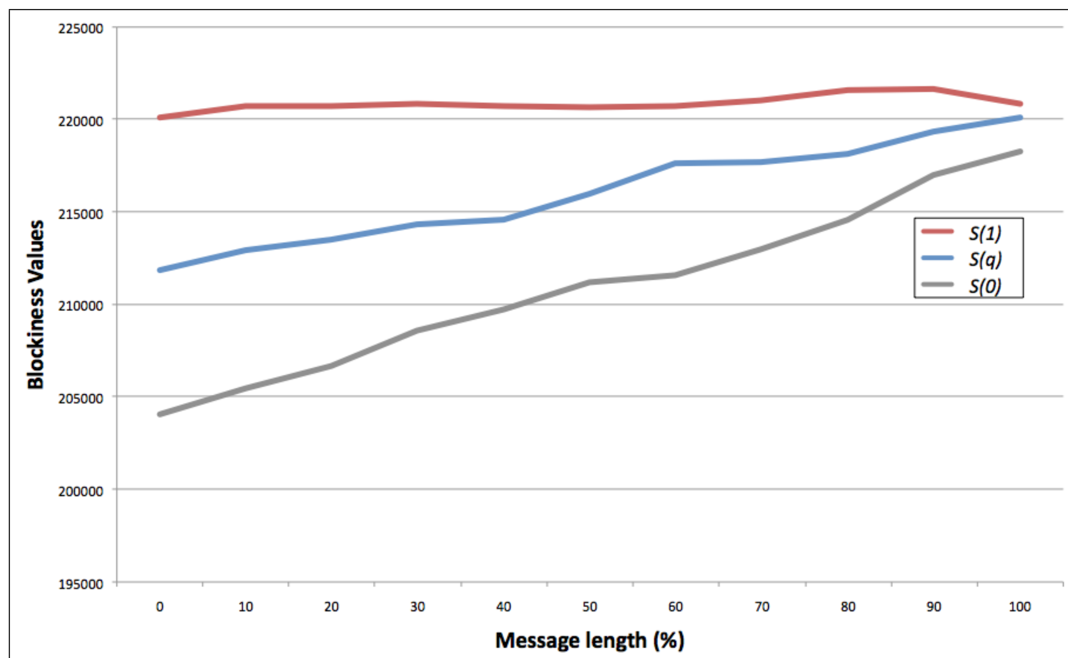


Figure 5.10: Plot of Blockiness values for $S(0)$, $S(q)$, and $S(1)$ for increasing q.

By running this investigation, it seems that we are able to prove that the suspect image $S(q)$ is a stegogramme as it falls between the plots of $S(0)$ and $S(1)$. The beauty of the attack is that we have been able to prove this via a blind approach, meaning the original cover image was never a part of the investigation.

5.4.4 Calculating The Message Length

By using the results we obtained from section 5.4.3, we can even go one step further and calculate the capacity of the embedded message q within the stegogramme. The method documented by Jessica Fridrich et al. in [6] uses *linear interpolation* to obtain a formula that can be used for calculating p. Linear interpolation makes it possible to derive the value of an unknown quantity when two other quantities are known [16]. In our case, we know that $q = 0$ for S_0, and $q = 1$ for S_1, so we can draw straight line between these two points. For any x that falls in between these points, we can use interpolation to work out its values. The standard formula for linear interpolation in terms of our example is shown in equation (5.3).

$$S = S_0 - q(S_0 - S_1). \tag{5.3}$$

By rearranging this formula in terms of p, we can obtain the formula shown in equation (5.4), and thus derive the embedding capacity of S_q.

$$p = \frac{S_0 - S}{S_0 - S_1} \tag{5.4}$$

5.5 Conclusion

The JPEG Calibration and Blockiness attack that we have seen in this chapter has proved that it is possible to determine whether or not a suspect image is a stegogramme, based entirely on its individual merits. That is to say that it is not necessary to obtain the original cover image in order for the attack to be successful.

The method is extremely accurate, with Jessica Fridrich reporting a 94% success rate in [5]. It is capable of not only detecting steganography, but also for deriving the embedding potential of stegogrammes. It has been tested thoroughly for both the *OutGuess 0.2* and *F5* algorithms, and has produced incredible results. Both of these algorithms function by ensuring the statistical properties of the stegogramme match that of the original cover image, so the fact that the procedure described in this chapter is capable of detecting embedding properties without the cover image makes this a very powerful tool for steganalysis.

However, there are some points that are worth noting about the assumptions the attacking method relies on. First and foremost, it relies on the choice of a good macroscopic quantity that behaves predictably as the message load is increased. If the macroscopic quantity is innaccurate, then so too will be the results yielded from the experiment. Secondly, it assumes that the macroscopic quantity behaves the same for the original cover image as is seen for the calibrated image. There may be cases where a macroscopic quantity behaves differently for the two images, in which case the results will be negatively affected. For the most part however, the calibrated image is proven to be an accurate model of the cover image, and so this should not prove to be a problem for the most part. It is mentioned in [5] that a nice development for the attack would be to automate the macroscopic quantities such that it reduces the chances of an incorrect decision being made.

The experiment we have discussed in this chapter was also carried out by Mario Leivaditis in [14], although that version of the experiment used the original cover image instead of the calibrated image. The results that we have obtained using a blind approach via the calibrated image are remarkably close to the results seen by Leivaditis. This proves that the calibration process is extremely accurate in emulating the cover image.

When using this method with a calibrated image, it is safe to assume that there will be a slightly higher ratio of false-positive evaluations. In other words, using an estimate of the cover image may often flag innocent images as stegogrammes; more so than if we used the original cover image. This is likely to be because the calibrated image has been cropped, resulting in a small yet noticeable change in the block structure that the Blockiness algorithm relies on. However, it is not usually so much of a problem in steganalysis that a few innocent images are flagged as stegogrammes. What would be far worse is if stegogrammes were considered innocent when they are not.

Finally, as with all the steganalytical techniques we have seen in this thesis, the chance of success is greatly reduced when the message load is close to zero. Obviously, the reason for this is that a smaller message involves few changes to the cover image to produce the stegogramme. Thus, the stegogramme will look remarkably similar to a clean image, even though it contains a hidden message. The calibration and blockiness attack is no different, and relies on a reasonably high embedding capacity to yield accurate results.

6 Research Conclusion

The research portion of this dissertation has succeeding in presenting a wide range of both steganographic and steganalytical techniques. We have been able to see the strengths and weaknesses of various stego-systems, not only from a steganographic viewpoint, but also in terms of how easy the artifacts of embedding can be spotted via steganalysis.

By researching both sides of the field in parallel, it has been interesting to note that a trade-off seems to exist. It seems to be the case that the easiest stego-systems to implement, are also the easiest to attack, whereas the more complicated stego-systems are much harder to attack. This of course makes perfect sense as the more complex systems are likely to be so because they embed the message data in a more intricate fashion than the simpler systems.

However, with that said, the simpler systems can still be used in such a way that they make life harder for the steganalyst, simply by embedding shorter messages. Short messages create a shorter bit-stream, which in turn requires less bit-flips to embed. With fewer modifications made to an image, it is much harder to spot a difference between the stegogramme and a clean version of the same image.

In terms of the steganalytical approaches discussed in this part of the dissertation, it is fair to say that the most useful approaches are the *blind* attacks that do not require the cover image to function. Whilst the *statistical* attacks are by far the most diserable attack for *targeted* steganalysis, they do rely on a great deal of knowledge about what we expect the cover image to look like. As these expectations are based on blind reasoning, they can often be incorrect which subsequently leads to a dramatic increase in false-negative classifications. *Blind* steganalysis on the other hand is much better because once we know we can mimic the cover image properties from a wide range of stegogrammes, we can be confident that the final classification will succeed. Much current research is being carried out for *blind* steganalysis for this reason, and all are proving to look very promising.

Part II

Software Development

7 Introduction

7.1 Overview

The software development portion of this project focusses on an implementation of most of the steganographic and steganalytical techniques described in Part I. This means that the end-product will provide a means for its users to embed a message within an image using one of several different steganographic algorithms, and then use various steganalytical techniques to identify the affect of embedding in these ways.

This chapter provides details of the aims and objectives of the development portion of the project, and also discusses the methodologies and design principles that were considered whilst building the system.

7.2 Disclaimer

Please be aware when reading the software development section of this project, that the idea of building a complete system was not considered until quite late on. The project is split between a research project and a development project, and as such, the development documentation that follows should not be read with an expectancy for acute attention to detail.

7.3 Intentions & Considerations

The end-product is intended to aid an education in the fields of steganography and steganalysis, but leans towards a steganalytical model that can be used to illustrate the affects of embedding message data within images. It will therefore need to make a clear distinction between the steganographic techniques and the steganalytical techniques, with careful attention paid to applying the correct terminology in all areas, such that the user obtains a clear understanding of the fields.

As the range of end-user can range from a complete novice to a more advanced user (in terms of their prior knowledge in the research area), the end-product will provide a Graphical User Interface (GUI) in order to accommodate all user types. In addition, back-end functions will be developed to accommodate users that wish to bypass the GUI. This will ensure that the system appeals to as many people as possible.

By developing the functions such that they are self-contained, it is possible that they can be used in association with an external bulk processing function in order to obtain results from a wide selection of source images very quickly. If this can be achieved successfully, the system will not only appeal to students who wish to learn more about

steganography and steganalysis, it will also mean that the tools are useful for steganalysis in a more active capacity. Subsequently, whilst the main focus was on producing a good user interface for each of the functions, attention was also paid to ensuring the longevity of the system as a whole, by way of developing the back-end functions such that they are all easy to use.

Also, by developing the functions in this manner, it means that new functions can easily be added that can operate alongside the existing functions. Thus, over time, the system has the potential to be highly desirable in the field of steganalysis.

7.4 Development Objectives

In accordance with section 7.3, the primary aim of this portion of the project is to create a useful tool to enable users to classify an image as either a stegogramme or a clean image. This can be done either by firstly creating a stegogramme using one of many built-in steganographic algorithms, or by using a suspect image obtained from external sources. The user should be able to test the image against a range of steganalytical techniques in order to make such a classification. This means that the system will need to offer a range of both steganographic and steganalytical activities.

As a secondary aim, an emphasis will be put on ensuring that the code is reusable. This will ensure that the tools are as useful as they possibly can be, as it will be possible for future developers to add new functions to the system, thus maintaining a central, up-to-date steganalysis tool.

7.5 Development Tools

The functions and GUI were developed using MATLAB (MATrix LABoratory), created by Mathworks. This development tool was chosen because it specialises in numerical computing. This means that there are many built-in library functions that allow the programmer to deal directly with the numerical components of a data source in many different ways (i.e. matrices, vectors etc.). This is exactly what steganography is concerned with. In steganography, we manipulate numerical values of an image such that they can contain message data.

In addition, the wide variety of mathematical functions available in MATLAB means that it is also possible to perform calculations on the image data in a very time-effcient manner. This is highly desireable when implementing the steganalytical methods described in Part I.

All of the functions were built in accordance with either the Image Processing toolbox, the JPEG[7] toolbox, or both. Subsequently, the functions can only be executed when both of these toolboxes are installed on the source terminal.

[7] The JPEG toolbox, developed by Phil Sallee, contains MATLAB routines for manipulating JPEG images. The toolbox can be downloaded from http://www.philsallee.com/jpegtbx/jpegtbx_1.4.tar.gz

8 API Documentation

8.1 Overview

In this chapter, we consider the system in terms of its Application Programming Interface (API). We discuss the logic behind the structure of the functions that comprise the end-product, explain what they do, and then describe how it is possible to develop them further if required.

We start by drawing upon the design principles that were considered when developing the functions, before moving on to discuss the functions in groups determined by their application. In the latter part of this chapter, we discuss how it might be possible to extend the functions for use with bulk processing applications by creating a wrapper function that calls each function as required. Finally, we close by suggesting how it is possible to create additional functions in such a way that they are compatible with the other functions previously developed.

8.2 Design Principles

As mentioned briefly earlier, the system was intended to be used for educational purposes, and thus incorporated a huge focus on the Graphical User Interface (GUI). However, it was also acknowledged that an alternative market might be for users that prefer to bypass the GUI and run the functions on a command line basis. For this reason, each of the functions for each separate group of operation (i.e. embedding, extracting, attacking) has been developed such that it takes the same parameters as inputs in the exact same order. The main benefit of this is that command line operations are made easier by the fact that the user only has to remember one structure for each of the tasks, and they will be able to use all of the related functions. We will discuss this in greater detail in sections 8.3 through 8.6.

The names of the functions have also been considered carefully so as to not add any confusion as to their operations. Abbreviations have been limited so that it is clear what each function does, and the functions have similar names where possible such that they can be predicted without needing to look them up. To illustrate an example of this, if we want to embed a message using the *JSteg* algorithm, the function we need is called *JSteg*. If we then want to extract the message data from the resulting stegogramme, we use the *extractJSteg* function. This is the same for all embedding and extracting functions such that if we want to use the *F4* algorithm to extract the message data, we can predictably use the function *extractF4*.

8.3 Embedding Functions

The embedding functions make it possible to embed a secret message (user specified) within an image by using one of several popular steganographic approaches. The resulting image is called a *stegogramme* and can effectively be sent to another user such that they can retrieve the message.

8.3.1 Function Names

The names of the embedding functions are listed below:

- *HideandSeek.m*
- *JSteg.m*
- *OutGuess01.m*
- *F3.m*
- *F4.m*

Each of the embedding functions operate according to the corresponding discussion in Chapter 3 of Part I. The *Hide & Seek* algorithm is slightly different from all of the other functions in that it can either be used for *sequential* or *randomised* embedding. The embedding type that it uses is reflected in the presence of a key. If a key exists, it will randomise the embedding approach, otherwise it will embed sequentially. All other functions will take a key as input and use it if it can, but there are some functions that do not require it.

8.3.2 Function Structure

To keep things as simple as possible, each embedding function has been structured in the same way such that it adheres to the following format:

```
function [stegogramme] = <embed_function> (image, message, key)
```

where *<embed_function>* should be replaced by the name of the embedding function. The three variables in parenthesis: *image*, *message*, and *key* are inputs to the function of type: *MxN (double)*, *1xN (char)*, and an *integer* respectively. The output *stegogramme* will be an *MxN (double)* image containing the message.

By adhering to this fixed structure, each algorithm can be called efficiently from the command line if necessary. This means that the user or developer only needs to understand the structure for one of the embedding functions in order to use them all.

The only input parameter that needs to be changed depending on the algorithm in question is the *image* parameter. This is because it depends upon which domain the algorithm works in. For spatial embedding, for example, the parameter will be an array of type double where each entry in the relay relates to the pixel value of that image. For

transform embedding however, the parameter will again be an array of type double, but each entry relates to a DCT coefficient of the image. We therefore need to ensure that we input the correct array for each algorithm.

The input parameter *key* is deliberately positioned at the end of the input list. This is because some of the algorithms require a key to shuffle the image data, and some do not. In MATLAB, if an input parameter is specified but never used, it will not produce an error. This means that it is not necessary to remember which algorithms require the *key* parameter, as it will not matter if it is provided when it is not needed.

8.4 Extracting Functions

The extracting functions are used in accordance with their embedding counterpart in order to extract the message that was embedded into the stegogramme.

8.4.1 Function Names

The names of the embedding functions are listed below:

- *extractHideandSeek.m*
- *extractJSteg.m*
- *extractOutGuess01.m*
- *extractF3.m*
- *extractF4.m*

Each extracting function operates in accordance with the pseudocode provided in Chapter 3 of Part I. They are effectively the inverse of the embedding functions such that they are capable of extracting the message data effectively.

8.4.2 Function Structure

As with the embedding functions, the extracting functions are also structured to adhere to a standard format. The format is as follows:

```
function [message] = <extract_function> (stegogramme, key)
```

where *<extract_function>* should be replaced by the name of the extracting function. The input parameter *stegogramme* is the data of the stegogramme produced from the embed function, and the *key* parameter is the exact same key that was given when embedding the message data. The output *message* is the message data that is retrieved from the stegogramme.

Obviously, the functions rely on the fact that the message was embedded using the corresponding embedding algorithm. For example, if *JSteg.m* was used to embed the

message, then *JSteg* will need to be used to extract the data correctly. Using the incorrect functions will result in an incorrect data stream coming out, thus it will appear as gibberish.

Again, the *key* parameter was deliberately positioned at the end of the input list such that it can be ignored if it is not necessary for extracting the message data.

8.5 Attack Functions

The attack functions are different to the embedding and extracting functions because each attack methodology is slightly different. This means that they cannot be structured such that they are fixed to a standard format, although every attempt has been made to ensure they are as close as possible.

8.5.1 Function Names

There are three methods of attack, the names of which are listed below:

- *VisualAttack.m*
- *HistogramAttack.m*
- *BlockinessAttack.m*

Each function operates as per the descriptions provided in Chapter 4 of Part I.

8.5.2 Function Structures

As the functions all work slightly differently, we should review each one separately so that we cause no confusion as to how it operates.

VisualAttack.m

The *VisualAttack.m* function can be used to plot the LSB plane of an image (in the spatial domain). The structure of the function is as follows:

```
function [bitplane] = VisualAttack(stegogramme, cover_image)
```

The function can take in either a single input parameter *stegogramme* or two input parameters if the cover image is available as well. If only the stegogramme is input, it will return the LSB plane of that image alone. If the cover image is also provided, it will show the LSB planes of both images, as well as showing an image of the difference between the two.

The output *bitpane* is an *MxN (double)* array of the LSB values derived from the stegogramme.

HistogramAttack.m

The *HistogramAttack.m* function can be used to return the histogram of an image (in the transform domain). The structure of the function is as follows:

```
function [h] = HistogramAttack(stegogramme, range)
```

The function requires two input parameters. The *stegogramme* parameter should be a *MxN (double)* array of quantised DCT coefficients, obtained via the JPEG toolbox. The second parameter *range (double)* refers literally to the range of values that fall across zero for the histogram. For example, specifying *12* will return histogram values that occupy the range -*12* to *12*.

The output parameter *h* refers to the histogram data.

BlockinessAttack.m

The *BlockinessAttack.m* function can be used to calculate the blockiness value of an image in either the spatial or transform domain. The structure of the function is as follows:

```
function [B] = BlockinessAttack(image)
```

The function requires a single input parameter *image* which is an *MxN (double)* array of either pixel values or quantised DCT coefficients.

The function will then calculate the Blockiness value of that image, and will return the result as *B (double)*.

8.6 Auxiliary Functions

There are a number of auxiliary functions that have been developed to make it easier to work with different data types. These include the ability to convert text to binary, binary to text, and also encrypt and decrypt a sequence according to the AES (Advanced Encryption Standard) algorithm[8].

8.6.1 Function Names

The auxiliary functions are named as:

- *text2bin.m*
- *bin2text.m*
- *aesencrypt.m*
- *aesdecrypt.m*

[8] AES encryption and decryption is taken from a pdf by Prof. Jörg J. Buchhol.
[source: http://buchholz.hs-bremen.de/aes/AES.pdf]

8.6.2 Function Structure

Each function operates slightly uniquely, so we should again discuss each function separately to avoid any potential confusion.

text2bin.m

The *text2bin.m* function makes it possible to convert string inputs to binary equivalents. The structure of the function is as follows:

```
function [msg] = text2bin(txt)
```

The function takes in one input parameter, *txt (char)* and converts this to a *8xN* binary string *msg (double)* by using ASCII as the basis for conversion.

bin2text.m

The *bin2text.m* function converts binary bit sequences to text. The structure of the function is as follows:

```
function [txt] = bin2text(msg)
```

The function takes in a single input parameter, *msg (double)* and converts it to a character string *txt (char)* by converting the binary string to ASCII integers, and then from ASCII integers to the equivalent characters.

aesencrypt.m

The function encrypts a message stream according to a key, by using the AES algorithm. The structure of the function is as follows:

```
function [ct] = aesencrypt(pt, key)
```

The function requires two input parameters. The first, *pt* is the message stream that is to be encrypted, and the second, *key (double)* acts as a passcode to unlock the encryption later on (see *aesdecrypt.m*).

The output *ct* is a 128-bit binary vector, hashed according to the MD5 algorithm.

aesdecrypt.m

The function decrypts a message stream that was previously encoded using the *aesencrypt.m* function, according to the same key. The structure of the function is as follows:

```
function [pt] = aesdecrypt(ct, key)
```

The function requires two input parameters. The first, *ct* is the encrypted message stream that was created using the *aesencrypt.m* function, and the second, *key (double)* is the same key that was used for encryption.

The output *pt* is the original message stream.

9 User Documentation

9.1 Overview

The graphical version of the steganography and steganalysis tools are designed to reinforce the user's previous knowledge of the concepts with real working examples. By using the GUI tools, the users can create their own stegogramme, extract the message from a stegogramme, and attack an image to find out if it contains message data.

This chapter describes the basic processes involved for achieving each of these tasks.

9.2 Requirements

In order to get started using the steganalytical tools, the user's should at least be familiar with the basic concepts of steganography and steganalysis. This can be achieved by reading Chapters 3 and 4 in Part I of this Dissertation. When user's are familiar with how the algorithms function on an operational level, they will be in a better position to relate to what they see in the GUI version of the tools.

Also, on a system level, user's will need to ensure that they have installed the Image Processing toolbox for MATLAB via the installation disk, and also installed Phil Sallee's JPEG toolbox from his website.

The interfaces have all been designed for use on a 13" laptop display as this is the dominant screen size for student user groups. Also, MATLAB does not resize the windows according to the monitor size, so it was much better to develop for a smaller screen size such that it will also be compatible with larger screen sizes. However, with that said, the interfaces have also been set up such that they open in the centre of any display.

9.3 Interface Arrangement

Each activity within the GUI can be located by following a series of menus that order them according to the domain in which they operate. The main menu offers four activities:

- *Create Stegogramme*
- *Extract Message*
- *Perform Steganalysis*
- *Calibrate Image*

Choosing either the *Create Stegogramme* or the *Extract Message* activities will present the user with a further menu that asks the domain that they wish to work in (i.e. *Spatial*, or *Transform*). Each steganographic algorithm (whether it is the embedding algorithm or the extracting algorithm) has been shelved according to the domain it operates in so, choosing *Spatial* will allow the user to select the *Hide & Seek* algorithm, and selecting the *Transform* option will allow the user to choose one of: *JSteg*, *OutGuess01*, *F3*, or *F4*.

Selecting the *Perform Steganalysis* option will again present a menu asking the user to select the domain for analysis. Choosing *Spatial* will bring up the *Visual Attack* interface, whilst choosing *Transform* will present a choice of either the *Histogram Attack* or the *Blockiness Attack*.

Finally, the *Calibrate Image* option simply brings up the *JPEG Calibration* interface.

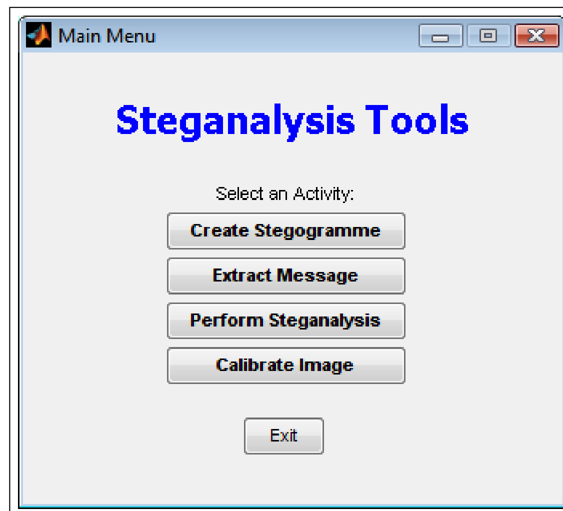


Figure 9.1: The Main Menu.

The activities themselves have been designed such that similar tasks and operations appear as similar as possible in terms of their layout. The reason for this is that it will allow the user to easily draw comparisons between the different steganographic or steganalytical strategies, and also improves the ease of use of the complete system. For example, the embedding interfaces all look similar, regardless of which algorithm is being called at the back end. Similarly, all of the extraction interfaces also look the same across the board.

9.4 Using The System

The system was built with the intentions of making things as intuitive as possible, such that the user is confident of what will happen when they click a button, before actually clicking it. However, for completion, we should discuss the major activities of the system such that the most novice user can determine how to operate each activity.

9.4.1 Loading the System

From the MATLAB command terminal, simply type *MainMenu* to load the system at the Main Menu.

9.4.2 Create A Stegogramme Using The *Hide & Seek* Algorithm

From the Main Menu, click *Create Stegogramme* and then *Spatial*. Finally click *Hide & Seek* to bring up the embedding interface.

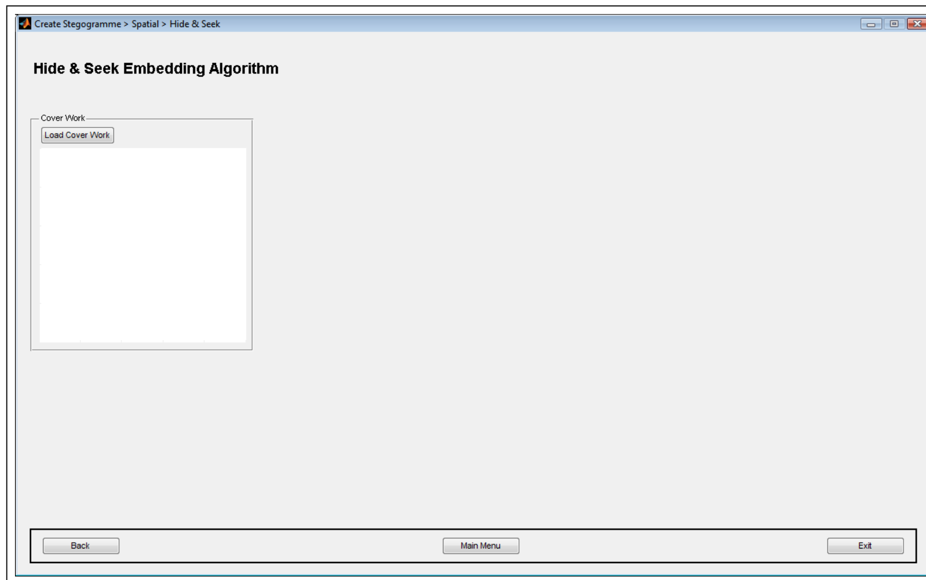


Figure 9.2: The *Hide & Seek* embedding interface.

Click the *Load Cover Work* button and browse for a PNG image to use as a cover image.

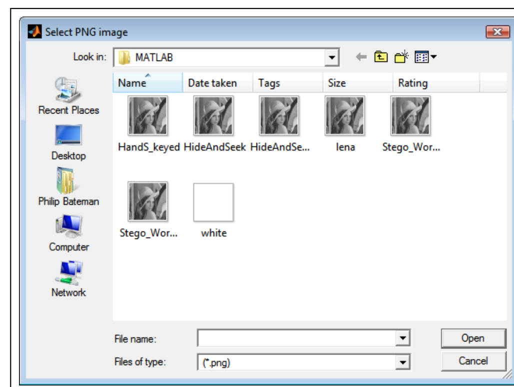


Figure 9.3: Browsing for a cover image.

Type a message in the text box that you wish to embed. Select AES Encryption if necessary, and enter a passphrase for decryption. Click *Done* to proceed.

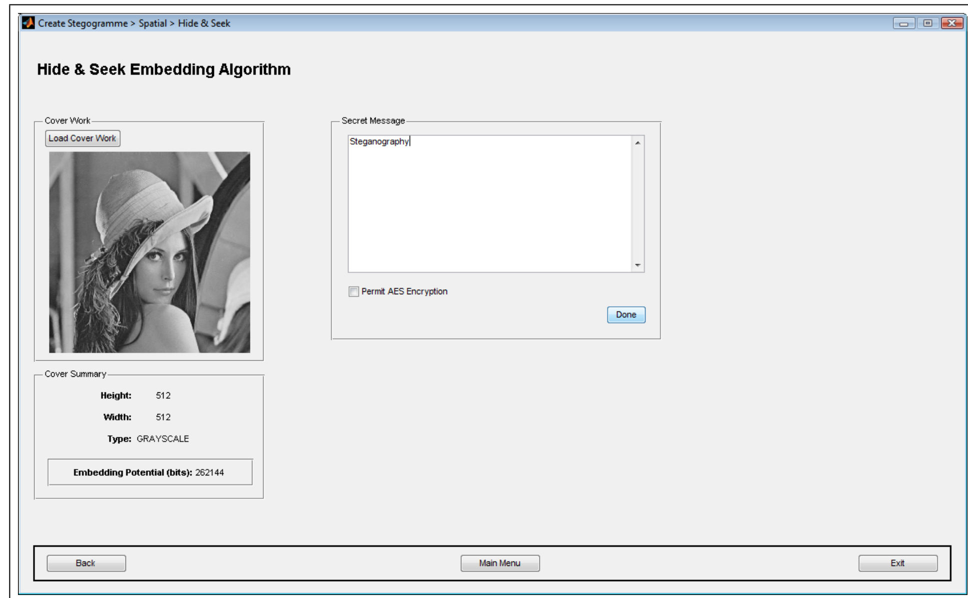
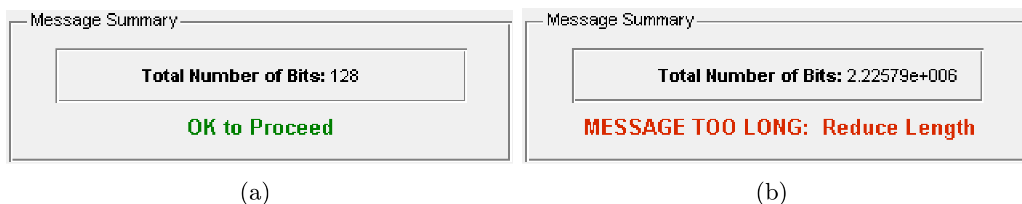


Figure 9.4: Adding a secret message.

The program checks that the length of the secret message does not exceed the maximum embedding potential of the image. The result will be displayed on screen.



(a) (b)

Figure 9.5: The results of the message check: (a) message length OK, and (b) message length too long.

If the message length is too long, then reduce its length and retry.

Otherwise, click the *Embed Message* button to embed the message. Note that you can tick the *Enable Randomised Embedding* check box to shuffle the image before embedding according to a PRNG seed. The same will need to be done when extracting the message using the *Hide & Seek* extracting algorithm within the *Extract Message* activity.

The stegogramme that was created is presented on the right-hand side of the screen.

Figure 9.6: Output of the *Hide & Seek* embedding interface.

The name and location of the stegogramme is also displayed, along with the embedding capacity of the image.

This same process can be followed for all embedding functions within the system as they are all designed in the same manner.

9.4.3 Extract Message Using The *Hide & Seek* Algorithm

Each extraction interface is extremely simple to use. Most of the extraction interfaces require only 2 clicks in order to retrieve the message data.

Simply start by clicking the *Load Stego Work* button to browse for the stegogramme that contains the hidden message.

Then click the *Extract Message* button to extract the hidden message. Be sure to tick the *Randomise before extracting* check box and enter the PRNG seed if a *randomised* approach to the *Hide & Seek* embedding algorithm was used. *Please note that all other embedding and extracting interfaces will only ask for the PRNG seed if it is needed.*

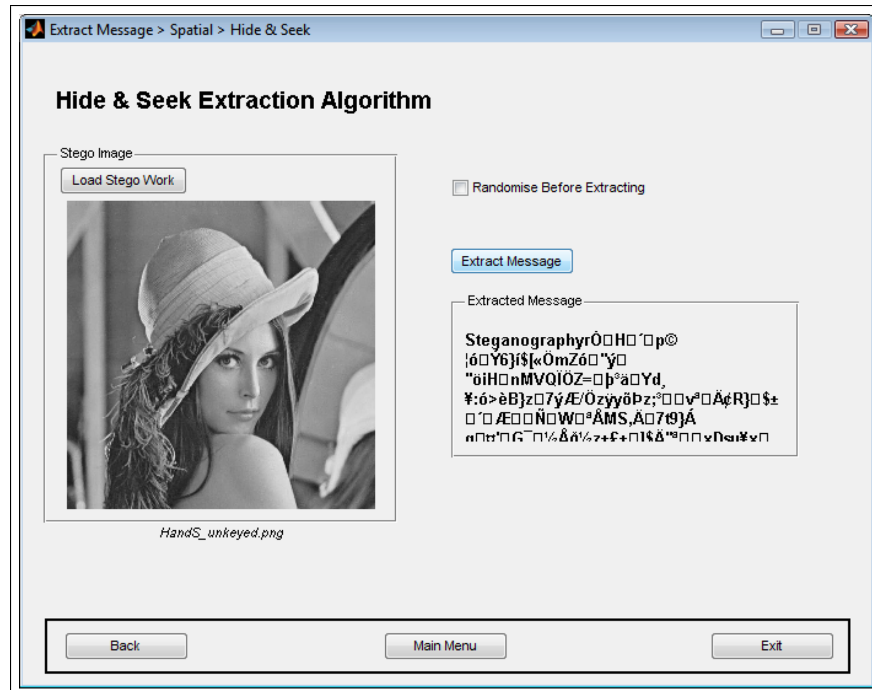


Figure 9.7: Output of the *Hide & Seek* extraction interface.

As the keys are simply used to determine whether or not a shuffle is required before embedding or extracting the message data, we have no way of knowing the length of the message when extracting the data. For this reason, the extraction interfaces are set to show the first 512 bits of the message data in the *Extracted Message* window.

Again, each extraction interface has been built such that the same layout and processes apply for all extraction functions.

9.4.4 Visual Attacks

The *Visual Attack* interface can be located in: *Perform Steganalysis >Spatial >Visual Attack.*

To use the attack simply browse for the suspect image by clicking the *Load Suspect Image* button, and click *Open.* If the cover image is also available, it can be selected by clicking the *Load Cover Work* button.

If both the suspect image and the cover work are given, both of their LSB planes will be presented under the thumbnail. In addition, the difference between the two bit-planes will be calculated and shown in a third window labelled *Difference.*

If only the suspect image is available, then the LSB plane of that image will be shown in isolation.

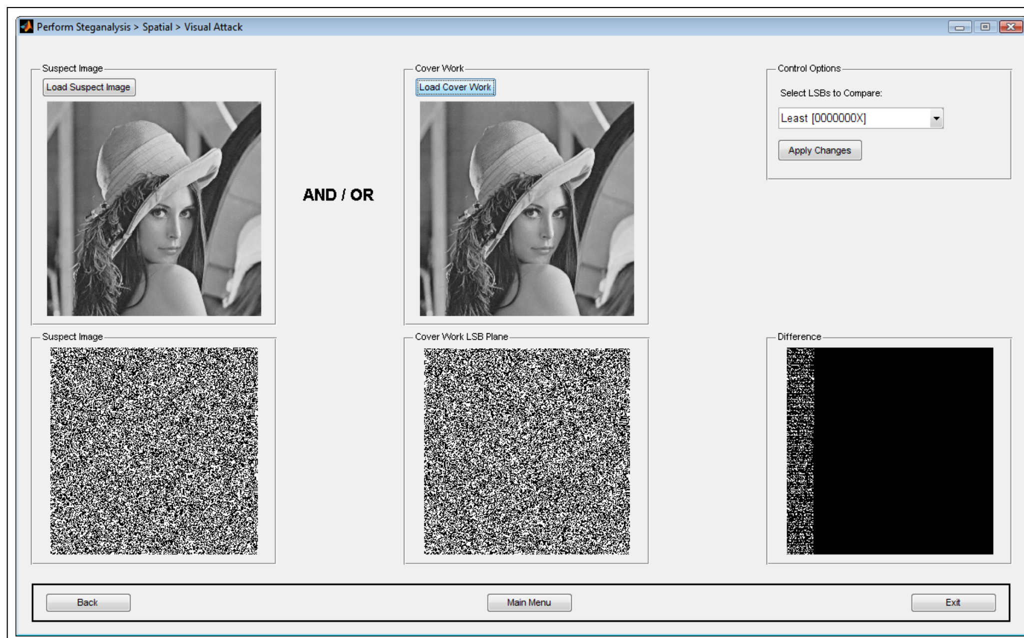


Figure 9.8: Output of the *Visual Attack* interface.

The bit plane to be examined can also be altered by selecting the appropriate entry in the *Control Options* panel.

9.4.5 Histogram Attacks

The *Histogram Attack* interface can be located by following the menus: *Perform Steganalysis >Transform >Histogram Attack*.

The interface is used by firstly browsing for the suspect image by clicking the *Load Suspect Image* button.

The *Histogram Attack Settings* panel can then be used to set up how you want the histogram to look. For example, you can choose the range of values, and whether to view the histogram data as a bar graph or a line graph.

If the cover work is available, then this can also be read into the interface such that plots can be made for both images, and a comparison can be made between the two resulting histograms.

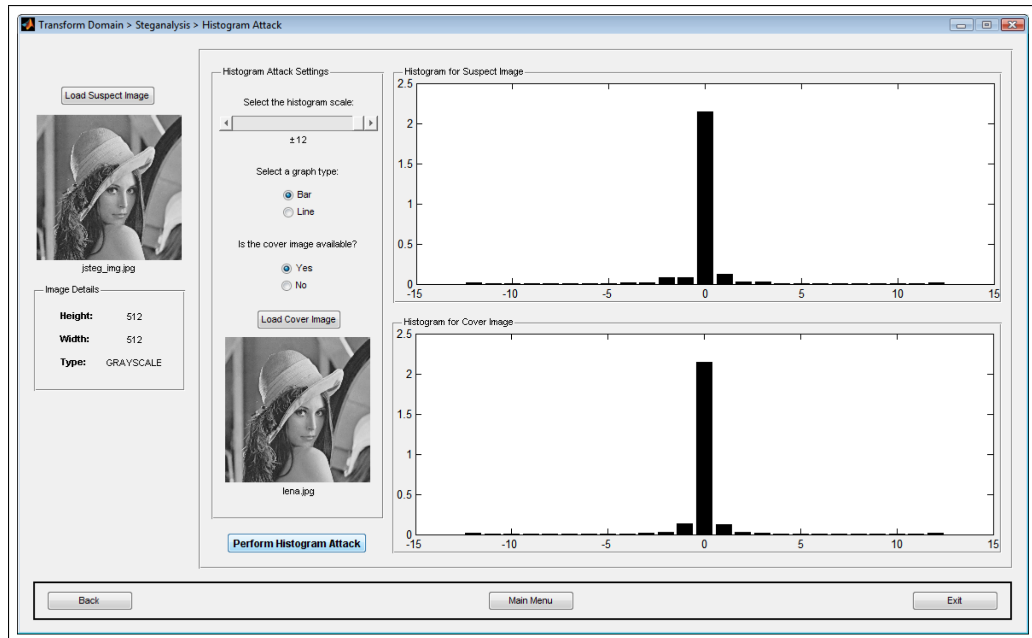


Figure 9.9: Output of the *Histogram Attack* interface.

9.4.6 Blockiness

The *Blockiness Attack* interface can be located by selecting: *Perform Steganalysis* >*Transform* >*Blockiness* from the *Main Menu*.

To perform the *Blockiness* attack, simply browse for the appropriate image by clicking the *Load Image* button.

Then calculate the *Blockiness* value of that image by clicking the *Calculate Blockiness* button.
The *Blockiness* value will be shown at the bottom of the window as per the example in Figure 9.10.

9.5 Navigation

At the bottom of each screen, there are three buttons that allow you to quickly access different parts of the system, or indeed to exit it completely. The buttons are labelled: *Back* for returning to the previous screen, *Main Menu* for returning to the *Main Menu*, and *Exit* for closing the system.

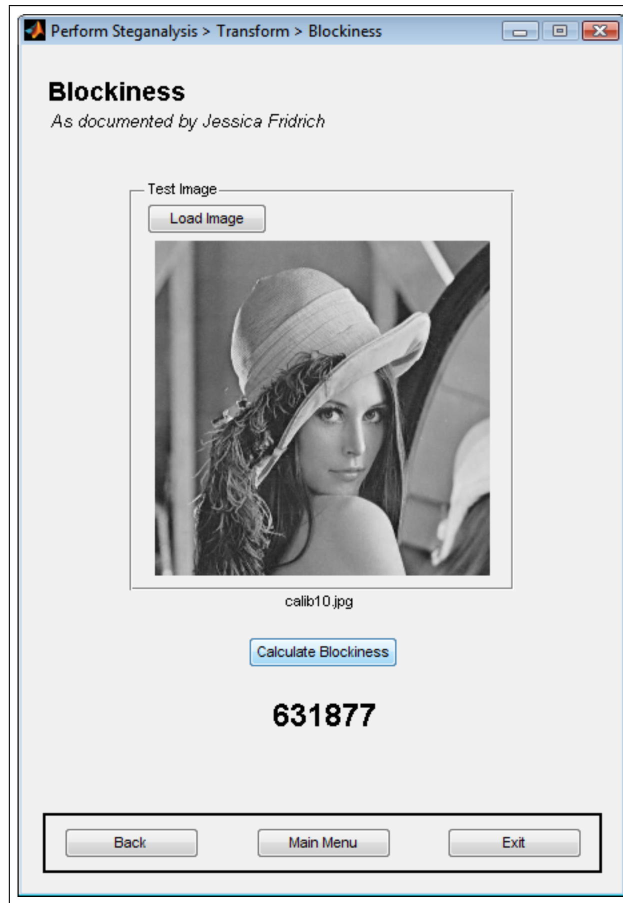


Figure 9.10: Output of the *Blockiness Attack* interface.

9.6 Adding New Functions

Regardless of the application for any new function that might be developed (i.e. steganographic embed/extract, or steganalytical), they can easily be included centrally will all of the other functions.

If any further embedding or extracting functions are produced, they should follow the same input/output structure that the current functions follow. Refer to sections 8.3 and 8.4 for further details.

The attacking functions should be limited such that they output a single parameter; the result of the attack. Feature vectors should be avoided as this will confuse the deliverables of the functions, and also make it harder to automate decisions via third party wrappers.

9.7 Bulk Processing

As the functions are all self-contained, they can potentially be processed in bulk to process data for a range of different purposes. This would be particularly useful to steganalysts wishing to run a particular attack on a large batch of suspect images, such that they can flag which images are likely to be stegogrammes.

In order to achieve this, a simple loop function could be developed such that it reads in x images, and runs the attack on each one. As the inputs and outputs are ordered the same for most functions, it is also possible to create new stegogrammes using the embed functions, and then attack them directly through the same bulk function.

10 Testing

10.1 Overview

Having produced all of the necessary functions for the system, it was necessary to test them such that we can be confident they are working as we expect. The steganographic functions for embedding and extracting message data can be easily tested at both bit level, and concept level to ensure the pixels are changed correctly, and that the message is extracted successfully. The steganalytical functions are much more subjective and therefore a little harder to test. However, through the research carried out in Chapter 4 of Part I, we can run sufficient tests to see that they are behaving as we expect.

This chapter discusses the testing methodology that was carried out for each of the functions that comprise the system.

10.2 Bit-level Testing

Each of the embedding functions operate slightly differently in terms of what they do to the image data in accordance with the message data that is being embedded. For example, some stego-systems leave the image data alone if we are embedding a 0 and increment the value if we are embedding a 1, whilst others will attempt to change the image data such that it matches the message bit stream. Subsequently, we can test each function by looking at the image data before and after embedding message data, to check that they are operating correctly.

For each test, we will embed the same message so that we can clearly see that the image data is changing as we expect across the functions. To keep things simple, we will embed the letter "S", which in binary is expressed as:

```
0 1 0 1 0 0 1 1
```

Using MATLABs debugging tool, we will be able to view the image arrays before and after embedding, and we will show the results as we progress.

10.2.1 *Hide & Seek* In *Sequential* Mode

Expected Behaviour

The LSB value of each pixel should be changed to match that of each entry in the message bit stream. To look at this more simply, a binary 0 in the message bit stream will create an even number by subtracting 1 from the current image data value (if it is not already

even). Similarly, a binary 1 will create an odd number by adding 1 to the current image data value (if it is not already odd).

Actual Behaviour

	1	2	3	4	5	6	7	8
1	162	162	161	161	161	160	160	160
2	162	162	162	162	161	161	161	160
3	163	163	163	162	162	161	161	161
4	163	163	163	163	162	162	161	161
5	163	163	163	162	162	161	161	161
6	162	162	161	161	161	160	160	160
7	161	161	160	160	159	159	159	159
8	160	160	159	159	159	158	158	158

(a)

	1	2	3	4	5	6	7	8
1	162	162	161	161	161	160	160	160
2	163	162	162	162	161	161	161	160
3	162	163	163	162	162	161	161	161
4	163	163	163	163	162	162	161	161
5	162	163	163	162	162	161	161	161
6	162	162	161	161	161	160	160	160
7	161	161	160	160	159	159	159	159
8	161	160	159	159	159	158	158	158

(b)

Figure 10.1: The pixel values (a) before, and (b) after embedding using *Hide & Seek* in *sequential* mode.

As we can see from the image data in (a), before the message was embedded, the first column of the image reads:

```
162    162    163    163    163    162    161    160
```

When we embed the first bit from the message stream (0) into the first pixel (162), there is no change as the LSB of the image value is already 0 (it is an even number). However, when we embed the second bit from the message stream (1) into the second pixel (162), a change is made such that the new LSB value of the image is 163. The is because the LSB of the pixel value before embedding was 0, and we needed it to be 1. The algorithm therefore increments the value such that the LSB value becomes 1.

By following the test for all the entries in the message stream, the same pattern emerges.

Success / Failure

Success.

10.2.2 *Hide & Seek* In *Randomised* Mode

Expected Behaviour

It is slightly harder to run the test when the image locations are scattered as we have no way of associating which message bit was embedded in which pixel. However, we can spot the areas that have changed and infer success or failure based on what we see.

We are expecting to see the same behaviour for this test that we saw for *sequential Hide & Seek*, the only difference is that the modified pixels will now be scattered over the image.

Actual Behaviour

	156	157	158	159	160	161	162	163
1	131	134	135	135	134	125	133	126
2	130	133	135	135	134	127	134	128
3	130	133	135	134	133	128	133	130
4	130	132	134	134	132	128	131	132
5	130	132	133	133	131	130	131	136
6	129	131	133	132	130	133	133	140
7	129	131	132	131	130	132	130	140
8	129	131	132	131	129	128	125	137

(a)

	156	157	158	159	160	161	162	163
1	131	134	135	135	134	125	133	126
2	130	133	135	135	134	127	134	128
3	130	133	135	134	133	128	133	130
4	131	132	134	134	132	128	131	132
5	130	132	133	133	131	130	131	136
6	129	131	133	132	130	133	133	140
7	129	131	132	131	130	132	130	140
8	129	131	132	131	129	128	125	137

(b)

Figure 10.2: The pixel values (a) before, and (b) after embedding using *Hide & Seek* in *randomised* mode.

We can see from Figure 10.2 that the entry in row 4 of column 156 changes after embedding. As the value increases by 1 we can infer that the message bit embedded at this location was a 1. By scanning the image for more changes, the same patterns emerged that we saw for *sequential Hide & Seek*.

Success / Failure

Success.

10.2.3 *JSteg*

Expected Behaviour

When embedding the message bits, we expect to see the following behaviour occur on the quantised DCT coefficients.

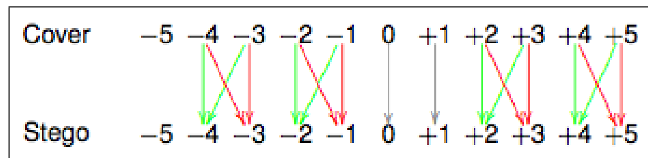


Figure 10.3: The expected bit flips from the *JSteg* algorithm. [21]

In addition, the DC coefficient, and any AC values equal to 0 or 1 should not be changed as a result of embedding.

Actual Behaviour

	1	2	3	4	5	6	7	8
1	130	5	3	0	0	0	-1	1
2	7	-1	1	-3	1	1	-1	1
3	-5	0	-1	1	0	0	0	0
4	2	1	1	0	0	0	0	0
5	-1	-1	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

(a)

	1	2	3	4	5	6	7	8
1	130	5	3	0	0	0	-1	1
2	6	-1	1	-3	1	1	-1	1
3	-5	0	-1	1	0	0	0	0
4	2	1	1	0	0	0	0	0
5	-1	-1	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

(b)

Figure 10.4: The pixel values (a) before, and (b) after embedding using the *JSteg* algorithm.

The first thing we can see from comparing image (a) with image (b) is that The DC value is not affected by the embedding process. The second coefficient value (7) is decremented by 1 (6) when we embed the first message bit (0). This is as we expect because the new value (6) now has an LSB that matches the message bit. The next coefficient (-5) remains unchanged as we embed a 1. This is also as we expect as the LSB value of this coefficient is already 1. The same pattern is true for the remainder of the message stream.

Success / Failure

Success

10.2.4 *OutGuess 0.1*

Expected Behaviour

The only aspect of the *OutGuess 0.1* algorithm that is changed from the *JSteg* algorithm is the fact that the manipulated coefficients are shuffled. Therefore, the same bit flips are expected for *OutGuess 0.1* that we saw for the *JSteg* algorithm.

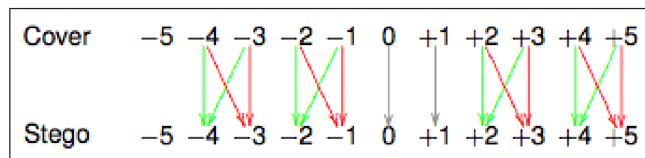


Figure 10.5: The expected bit flips from the *OutGuess 0.1* algorithm. [21]

We will however have the same difficulty running the test for this algorithm that we saw for the *randomised Hide & Seek* algorithm, because we again have no means of knowing which value was changed as a result of embedding which message bit. We will therefore have to spot the values that change, and check that they have changed as per our expectations.

Actual Behaviour

As we can see from image (a) in Figure 10.6, the AC coefficient in row 35, column 121 is -3 before embedding, and is changed to -4 in image(b) after embedding. This is what we would expect to see if we are embedding a 0, as the LSB of the coefficient has been changed from 1 to 0 after embedding. By scanning the image more thoroughly, more of the same patterns were witnessed, so we can be confident that the bit flips are working correctly.

	121	122	123	124	125	126	127	128
33	-22	-11	4	4	0	0	0	-1
34	7	6	3	1	-2	-1	0	0
35	-3	-1	3	-1	0	0	-1	0
36	-1	-2	-2	1	0	0	1	0
37	-1	1	0	0	0	0	0	0
38	1	0	0	0	0	0	0	0
39	0	0	0	0	0	1	0	0
40	0	0	0	0	0	0	0	0

(a)

	121	122	123	124	125	126	127	128
33	-22	-11	4	4	0	0	0	-1
34	7	6	3	1	-2	-1	0	0
35	-4	-1	3	-1	0	0	-1	0
36	-1	-2	-2	1	0	0	1	0
37	-1	1	0	0	0	0	0	0
38	1	0	0	0	0	0	0	0
39	0	0	0	0	0	1	0	0
40	0	0	0	0	0	0	0	0

(b)

Figure 10.6: The pixel values (a) before, and (b) after embedding using the *OutGuess 0.1* algorithm.

Success / Failure

Success

10.2.5 *F3*

Expected Behaviour

When embedding the message bits according to the *F3* algorithm, we expect to see the following bit flips occur on the quantised DCT coefficients.

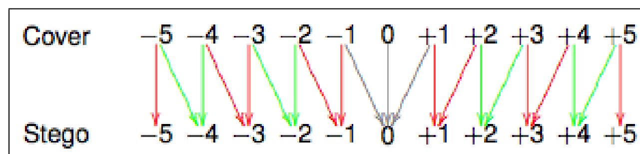


Figure 10.7: The expected bit flips from the *F3* algorithm. [21]

Note that we do not embed on the DC values, nor the AC coefficients equal to 0. However, this time around, we can embed on coefficients equal to 1 or -1. However, if

they produce a 0 then we must re-embed the same message bit until it produces something other than a 0.

Actual Behaviour

	1	2	3	4	5	6	7	8
1	130	5	3	0	0	0	-1	1
2	7	-1	1	-3	1	1	-1	1
3	-5	0	-1	1	0	0	0	0
4	2	1	1	0	0	0	0	0
5	-1	-1	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

(a)

	1	2	3	4	5	6	7	8
1	130	5	3	0	0	0	-1	1
2	6	-1	1	-3	1	1	-1	1
3	-5	0	-1	1	0	0	0	0
4	2	1	1	0	0	0	0	0
5	-1	-1	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

(b)

Figure 10.8: The pixel values (a) before, and (b) after embedding using the *F3* algorithm.

We can see from both image's that the DC values are definitely ignored, as the value remains the same after embedding.

The first coefficient that changes can be seen on row 2, column 1. The coefficient was 7 before embedding and it is changed to 6 when embedding a 0. This is to be expected as Figure 10.7 shows that the coefficient values >0 are either kept the same or decremented by 1. As the LSB of 7 is 1, the value was decremented such that it becomes 0. The next coefficient (-5) remains the same as we are embedding a 1 and the LSB of -5 is already 1. The same pattern can be seen for the length of the message stream.

Success / Failure

Success.

10.2.6 *F4*

Expected Behaviour

We expect to note the following behaviour on the quantised DCT coefficients when embedding the message data according to the *F4* algorithm.

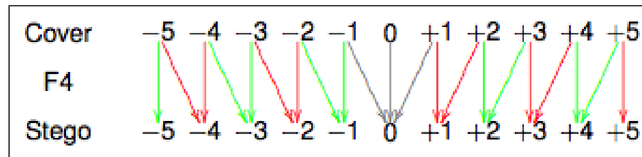


Figure 10.9: The expected bit flips from the *F4* algorithm. [21]

Again, we do not embed on the DC coefficients or any AC coefficient equal to zero. Also, as we saw for the *F3* algorithm, we can embed on AC coefficients equal to -1 and 1, but must re-embed the message bit if a zero value is produced.

Actual Behaviour

	1	2	3	4	5	6	7	8
1	130	5	3	0	0	0	-1	1
2	7	-1	1	-3	1	1	-1	1
3	-5	0	-1	1	0	0	0	0
4	2	1	1	0	0	0	0	0
5	-1	-1	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

(a)

	1	2	3	4	5	6	7	8
1	130	5	3	0	0	0	-1	1
2	6	-1	1	-3	1	1	-1	1
3	-4	0	-1	1	0	0	0	0
4	2	1	1	0	0	0	0	0
5	0	-1	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

(b)

Figure 10.10: The pixel values (a) before, and (b) after embedding using the *F4* algorithm.

Again, the DC coefficient is the same for both image (a) and image (b) meaning that the algorithm correctly avoids embedding on these values. In addition, we can see that the second AC coefficient in image (a) (7) is correctly decremented to 6 when embedding a 0. Similarly, we see that the third AC coefficient (-5) increments to -4 when a 1 is embedded. This is as we expect from the bit-flips denoted in Figure 10.9.

Success / Failure

Success

10.2.7 Message-level Testing

As the purpose of image steganography is to send a secret message to somebody by hiding it within an image, we should also test that we can retrieve the message successfully.

The test was carried out such that a message was embedded for every embedding function. The extracting function was then run to see what message was extracted. The results are shown in the following Figure.

Embedding Algorithm	Message Embedded	Message Extracted
Hide & Seek (sequential)	*Hide and Seek Message*	*Hide and Seek Message*
Hide & Seek (randomised)	*Hide and Seek Message*	*Hide and Seek Message*
JSteg	*JSteg Message*	*JSteg Message*
OutGuess 0.1	*OutGuess01 Message*	*OutGuess01 Message*
F3	*F3 Message*	*F3 Message*
F4	*F4 Message*	*F4 Message*

Figure 10.11: Test results for embedding and extracting for all steganographic functions.

As Figure 10.11 shows, the message was successfully extracted from each stegogramme. This, coupled with the results from the bit-level testing, means that we can be confident that the steganographic functions within the system operate as we expect.

10.3 Attack Testing

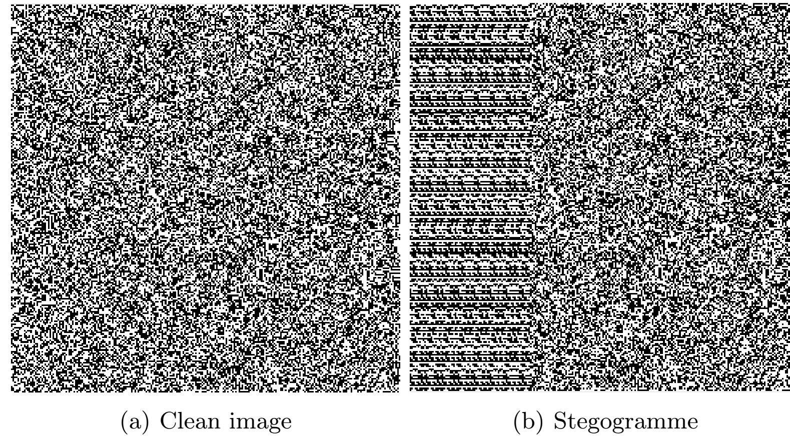
As the attack functions do not make automatic classifications with regards to whether or not an image is a stegogramme, the method for testing the functions depends on viewing the results and ensuring that they behave differently for *clean* and *dirty* images.

We will therefore test each function separately and ensure that we yield the correct results for clean images and stegogrammes.

10.3.1 Visual Attacks

To test the Visual Attack function, a stegogramme was created using *sequential Hide & Seek* to embed a message at roughly 30% capacity. The LSB planes were then plotted

for a clean image and the stegogramme. Figure 10.12 shows the results from this test:



(a) Clean image (b) Stegogramme

Figure 10.12: The LSB planes of a clean image and a stegogramme.

As we can see, the LSB plane for the clean image is notably distorted in all areas. However, the LSB plane of the stegogramme clearly shows the traces of embedding on the left-most region of the image.

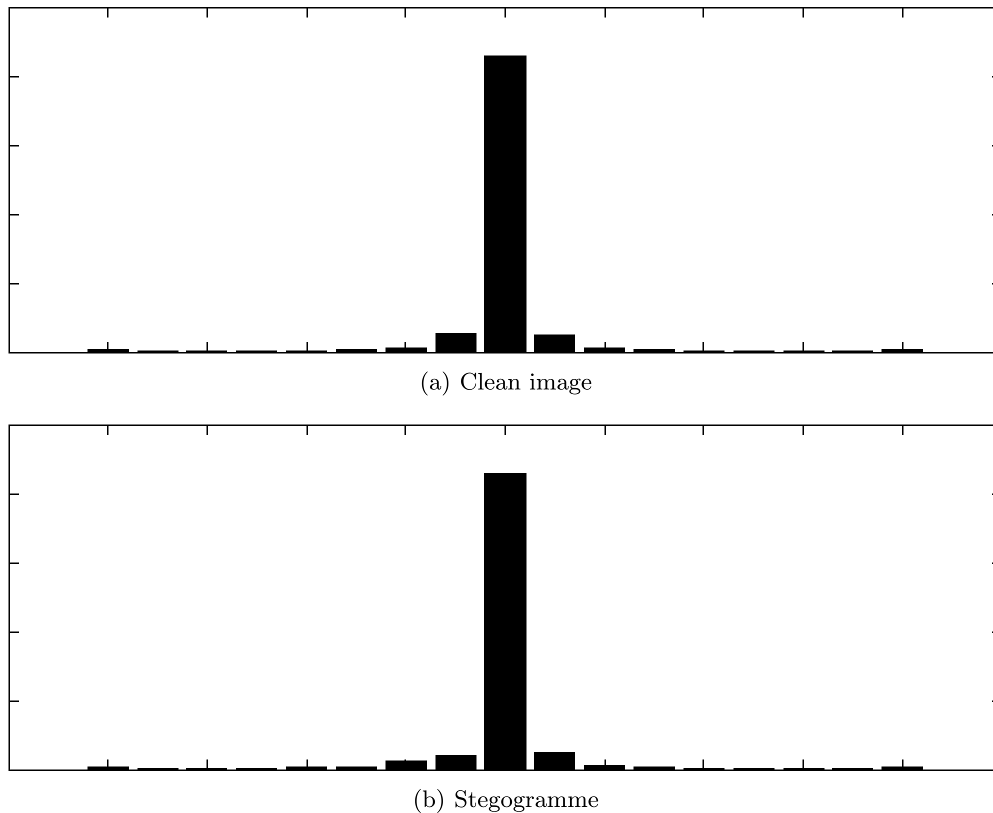
This proves that the attack function is capable of displaying the correct information for both a clean and distorted image.

10.3.2 Histogram Attacks

In order to test the *Histogram Attack* function, a stegogramme was created using the *JSteg* algorithm at roughly 50% capacity. The histogram of the resulting stegogramme was plotted along with the histogram for a clean image. The results can be seen in Figure 10.13.

The histogram in image (a) shows a fairly symmetric distribution across the centre, whilst image (b) illustrates a more jagged structure. As discussed in Chapter 4 in Part I, this is to be expected as the bit-flipping procedure of the *JSteg* algorithm tends to force the frequencies of odd values to be higher than the frequencies of even values.

As we can clearly note the traits of embedding on a stegogramme, and also see the symmetric structure of a histogram for a clean image, we can be confident that this attack is capable of identifying stegogrammes.



(a) Clean image

(b) Stegogramme

Figure 10.13: The histograms of a clean image and a stegogramme.

10.3.3 Blockiness Attack

The *Blockiness Attack* is difficult to test as the output does not give anything away as to whether or not the image is clean or a stegogramme. However, from what we discussed in Chapter 5 of Part I, we should notice that the Blockiness value is larger for stegogrammes than it is for clean images, as a direct impact of transform domain embedding.

In order to test the function, a stegogramme was created using the *OutGuess 0.1* algorithm at roughly 50% capacity. The Blockiness value was then calculated for this image and a clean image such that a comparison could be made. The yielded results were as follows.

```
clean image = 216677

stegogramme = 236906
```

The test clearly shows that the Blockiness value for a stegogramme is higher than that of a clean image, and this is a good indication that the function is performing as we would expect.

10.4 JPEG Calibration

The *JPEG Calibration* function cannot really be tested on a system level. However, its purpose is to mimic the statistical properties of the cover work, which means we can test that the function works correctly by plotting the histogram of a cover image against the histogram of the calibrated image. This can be done by creating a stegogramme using the *OutGuess 0.2* algorithm, to create the calibrated image, and plotting the histogram against the image that was used to produce the stegogramme. If both histograms are roughly the same, then we know that the function is working correctly.

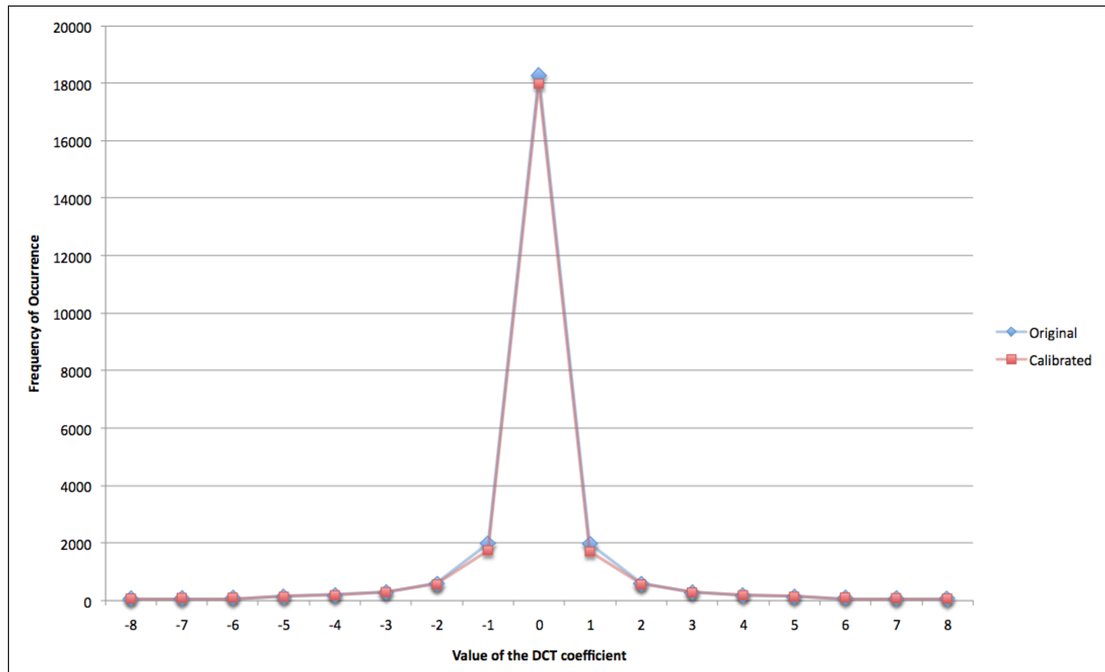


Figure 10.14: The histograms of a clean image and the calibrated image.

As we can see from Figure 10.14, the histogram for the calibrated image closely matches that of the original cover image. This is in accordance with our expectations, so we can be confident that the function is working properly.

10.5 Summary

The tests that have been carried out in this section have provided us with enough information to be confident that the functions work correctly. We have thoroughly tested the steganographic functions both on a bit-level and also on a message-level, and we have proved that they all work to our expectations. It would have been more useful to cross-test these functions by embedding a message, and then attempting to decode them

using the packages developed by the authors, and vice versa. This would mean that we not only know that they are working as we expect, it would also prove that they can work in association with the functions approved by the original authors. However, this was not possible as only the *JSteg* package could be located, and some difficulties were experienced trying to install it on the test computer. With that said, the test strategy outlined in this chapter still gives us a high level of confidence that the embedding and extraction functions work correctly.

Having also tested the attack functions and the *JPEG Calibration* function, we can be even more confident that the functions are behaving as we expect.