Image Steganography

Henrik Brüsecke
B.A.(Mod.) Computer Science
Final Year Project  April 2015
Supervisor: Dr. Hitesh Tewari

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Acknowledgements

Many thanks to my supervisor Dr. Hitesh Tewari for his helpful advice and guidance throughout the project. Thanks for leaving the project relatively open-ended and allowing me to investigate the effectiveness of cloud steganography.

I would also like to thank my family for their support and help in attempting to visually detect hidden clouds.
Original project specification

**Image Steganography**

The techniques for secret hiding of messages in an otherwise innocent looking carrier message belong to the field of steganography. The purpose of steganography is to conceal the very presence of secret information. To make the communication more secure, the secret information can be compressed and encrypted before it is hidden in the carrier. In this project we would like to explore the field of steganography and implement a mobile based application.
Abstract

Initially the aim of the project was to implement a mobile based image steganography application. The objective of the project was changed to explore the feasibility of cloud steganography.

Image steganography attempts to hide secret data in images. Most steganography techniques attempt to exploit redundancies and/or excessive accuracy in image formats. Adding a payload to an image usually changes the cover image’s statistical footprint. Images generally contain patterns and aren’t random whereas compressed and/or encrypted payloads usually are random. Multiple techniques exist to try and minimise the statistical impact. For example, one method is to spread the payload evenly across the entire image.

This project explores disguising the payload as a naturally random entity, such as a cloud, crack in a road, grass or similar. Essentially the data is hidden in plain sight. Natural phenomena exhibit features that are random, such as the shape of clouds. If something that is statistically random is manipulated in a random way then there is no way of telling that anything has changed. This fact makes cloud steganography very resilient to statistical attacks by principal.

A cloud steganography application was created. The implementation still has many flaws. The current implementation could be a very secure steganography application if the flaws are removed.
Contents

Chapter 1 – Introduction
- Image steganography problems
- Statistical resilience
- Chapter guide

Chapter 2 – State of the art
- LSB
- JPEG
- Other techniques

Chapter 3 – Proposed solution

Chapter 4 – Implementation
- Introduction
- Technologies used
- Implementation stages
- PAQ8PX compression
- AES encryption
- Crude cloud generation
- Blur
- Correction
- Decoding

Chapter 5 – Steganalysis

Chapter 6 – Conclusions

Appendix – How to use

References

DVD – Includes application, source code and sample image
Chapter 1 – Introduction

The field of steganography is dedicated to the science of hiding information in other information, such as text, images or video. The main aim is to prevent everyone except for the intended recipient to become aware of the hidden information. This is different to encryption where the objective is to prevent anyone except for authorised people to be able to make sense of the information. Encryption and steganography can complement each other. Steganography is particularly useful in situations where sending an encrypted file would arouse suspicion and may attract unwanted attention.

For example, as an undercover tourist in North Korea it would be much safer to send an image of the North Korean landscape with steganography applied to the picture than to send an encrypted file. The secret information would be hidden in the image. Authorities probably wouldn’t suspect anything if a picture of the North Korean landscape was sent out of the country. Had the undercover tourist sent an encrypted file then in the worst case scenario authorities would have forced the suspect to give up the encryption key.

Steganography isn’t a new practice and has been used throughout history ranging back to the ancient Greeks. As a matter of fact, the word steganography originates from ancient Greece and literally means “covered writing”. One of the earliest recorded uses of steganography dates back to 440 B.C. when the Greek historian Herodotus recorded how Histiaeus shaved the head of one of his slaves and wrote a message on the slave’s scalp [1]. After the slave’s hair had grown back the message was hidden. The information could only be extracted by shaving the slave’s head again.

Image steganography is concerned with hiding information in images specifically. Pictures are very popular carrier files as they often contain some redundancy and are widely used and exchanged on the internet [2]. Furthermore a wide range digital image formats make use of excessive accuracy. When this accuracy is reduced it usually isn’t visible to the human eye. Some steganography techniques only work with specific image formats as the compression schemes employed vary significantly. Image compression schemes can be categorised as being either lossy or lossless. Lossless compression guarantees that a lossless compressed image will be an exact replica of the original image once decoded. Lossy compression usually achieves much better compression ratios at the expense of losing some information. Lossy compression aims to only remove information in an image that isn’t visible to the human eye.
Image steganography problems:

There are two main ways of detecting whether an image has been tampered with. The first method is to try and detect changes in the visual domain. Most steganography techniques attempt to avoid changes to the image becoming visible to the human eye. The second method relies on investigating the statistics of the image and attempting to detect statistical inconsistencies within the image.

Making changes to an image invisible or attempting to conceal that an image has been tampered with is an obvious requirement of image steganography. According to T. Morkel, J.H.P. Eloff and M.S. Olivier, “The moment that one can see that an image has been tampered with, the algorithm is compromised.”[2]

Example of a steganography image with extremely poor invisibility:

The human eye can easily detect that this image has been tampered with, defeating the purpose of steganography.

Using statistical steganalysis tools one can detect that an image has been tampered with. These tools look for statistical inconsistencies within the cover image. No matter what steganography technique you use, essentially bits are going to be flipped. It is impossible to embed data within an image without either adding bits or flipping them. The only exception to this rule is when the cover
image already has the right bit configuration in the places where the payload is hidden. This scenario is extremely unlikely unless the payload is very small.

As a consequence of changing bits in the cover image the statistical footprint will change. Usually the payload is statistically random due to compression and encryption. It is almost always necessary to compress large payloads as the capacity of most steganography techniques is very limited. Usually there is a trade-off between payload capacity and how easily one can detect the steganography.

Images, on the other hand usually contain patterns and don’t exhibit statistically random footprints. A completely random image would consist exclusively of noise and if it was grayscale would look like the image on a TV that isn’t tuned properly.

This is what a random image looks like: (Generated using modified version of cloud steganography application)

Statistically random images generally can’t be compressed as there aren’t any redundancies to exploit.
**Statistical resilience:**

Multiple techniques exist that attempt to minimise the statistical impact of steganography on cover images.

The simplest way to reduce the statistical impact is to reduce the size of the payload. Even very simple image steganography techniques become very secure when the payload is tiny relative to the size of the cover image. When the payload size to cover image size ratio is very low then it is practically impossible to determine if there is a hidden payload in the cover image using steganalysis, especially when the payload is spread across the entire image. Compression can be used to reduce the size of the payload. The better the compression that is used, and subsequently the smaller the payload size, the less likely it becomes that the steganography can be detected by an attacker.

Steganalysis techniques such as the LSB detection technique proposed by Jessica Fridrich and Miroslav Goljan can’t reliably detect LSB embedding when less than 1% of the pixels in the cover image have been modified.[3]

Another common approach to minimise the statistical impact is to spread the payload evenly across the cover image instead of embedding it sequentially from the start of the image. Usually a key determines which pixels are chosen at random for hosting the payload. Using a key has the added advantage that nobody can extract the payload from the cover image unless the extractor knows the key. Even if an attacker becomes aware that an image has been tampered with, the key ensures that the attacker won’t be able to extract hidden data, even when the attacker is aware of the payload’s presence.

Edge-based image steganography offers a higher statistical resilience than LSB embedding.[4] This is because in edge-based image steganography only edge pixels are used to embed the payload. Edge pixels are more random than non-edge pixels as the colour of an edge pixel depends on what percentage of each object in that pixel was present when the image was taken. For example, imagine a simple image containing only a field of grass and a sky. A pixel that is located within the transition from sky to field will contain both the sky and field. The LSB of an edge pixel is fairly random as the pixel is within an area of abrupt change.
Chapter guide:

Chapter 1 Introduction.

Chapter 2 describes current steganography techniques.

Chapter 3 explains in detail the proposed solution.

Chapter 4 explains in detail what was implemented for this project.

Chapter 5 explains how well the current implementation performs.

Chapter 6 discusses what could be done to improve the current implementation.
Chapter 2 – State of the art

This chapter gives an overview of image steganography techniques, from basic to more advanced methods.

**LSB:**

The LSB technique hides the payload in the least significant bits of each colour channel in each pixel. This results in a capacity of 3 bits per pixel in a 3 channel colour scheme such as RGB or HLS. Grayscale images would only provide a capacity of 1 bit per pixel as there is only one colour channel. If a 24 bit RGB colour scheme is used, 8 bits per channel, then flipping the least significant bit in each channel has only a very small effect on the overall colour. These changes usually aren’t visible to the human eye. Sometimes it is even possible to modify the last two or three least significant bits without the changes becoming visible to the human eye.

On average only every second bit needs to be flipped as there is a 50% chance that the bit already has the correct value. In extreme cases, it may not be necessary to flip any bits at all, however this is very unlikely. LSB steganography only works reliably when used with lossless image formats such as BMP or PNG. Lossy compression such as JPEG could result in a partial loss of the payload.

LSB image steganography boasts a high payload capacity, 1/8 of the cover image, or more if the last two or three least significant bits are used. Another key advantage of the LSB method is it’s invisibility to the human eye. It is visually practically undetectable. However, LSB has one major weakness, it is very easy to detect using steganalysis tools as it leaves a statistical footprint. The least significant bits of most images aren’t random, whereas the payload usually is.

More advanced LSB algorithms only manipulate random bytes in an image, effectively spreading the payload across the entire image. A key determines which bytes are selected for manipulation. Key based LSB only provides more statistical resilience when the payload doesn’t just barely fit into the image, i.e. when it is necessary to manipulate all or almost every byte in the image to fit the payload. However, a further advantage of key based LSB is that even when an attacker detects the steganography, it won’t be possible to extract the actual information in the correct order without knowing the key.
JPEG:

JPEG (Joint Photographic Experts Group) is an image format that uses lossy compression. Using simple LSB embedding in JPEG images generally doesn’t work as some of the payload would be lost.

JPEG compression can be categorised into different stages. Some of these stages are lossy, others are lossless. First, the colour scheme of the image to be compressed is converted to YUV. Next the two colour components, U and V, are quantized to reduce the file size. The quantization results in the loss of some colour accuracy, i.e. it is a lossy stage. Next, the image is split into 8x8 blocks and the DCT transform is applied. The resulting coefficients are quantized, again some information is lost here. The final stage consists of simple lossless Huffman encoding.

One type of JPEG steganography modifies the least significant bits of the quantized DCT coefficients.[7] This is safe to do as the only compression stage after this, the Huffman encoding, is lossless.

According to T. Morkel, J.H.P. Eloff and M.S. Olivier, JPEG steganography provides high invisibility and is more robust to statistical attacks than primitive LSB applied to lossless image formats.[2] A disadvantage is the reduced capacity, however this is due to the fact that JPEG cover images are generally smaller in size than images compressed using lossless compression.

Other techniques:

A lot more image steganography techniques exist that aren’t included in this chapter. Other methods include spread spectrum steganography, edge based steganography, pixel intensity steganography, histogram shifting steganography and texture based steganography.[8]

According to T. Morkel, J.H.P. Eloff and M.S. Olivier, one of the safest techniques is spread spectrum steganography.[2] It is invisible to the human eye and is practically undetectable using statistical techniques. In spread spectrum steganography, a narrowband signal is spread across a much wider band of frequencies. According to Lisa M. Marvel, Charles G. Boncelet, Jr. and Charles T. Retter, “After spreading, the energy of the narrowband signal in any one frequency band is low and therefore difficult to detect.”[9]
Chapter 3 – Proposed solution

Instead of re-implementing an existing solution this project is an attempt to test a different kind of approach to mitigating statistical impact on cover images. Rather than modifying existing pixels in a discreet way and hoping that the changes are subtle enough to go undetected, an object with some naturally random properties is inserted into the cover image. This naturally random object represents the payload.

Many phenomena that have some random properties are regularly featured in pictures. Examples are clouds, grass, waves on water, leaves on a tree, cracks on a road, gravel and rocks. Only some properties of these entities are random. The arrangement of small rocks in a field of gravel is random. By re-arranging the positions of each rock in the gravel field a secret message can be embedded. For this project clouds were used as payload hosts as they were deemed easier to draw than a field of gravel or grass. Furthermore, clouds are very common in images in general. Almost every outdoor picture features a sky with clouds.

The shape of clouds are, to a degree, naturally random. The edges of clouds can be used to embed information. To hide the data in an actual image one would either have to modify the edges of an existing cloud or render a synthetic cloud into the cover image. The second option was chosen for this project as it carries less risks. If an existing cloud in the cover image is modified and happens to become smaller than the original cloud then one would have to fill in the pixels that used to be part of the cloud with sky. What the sky would look like behind the cloud can only be guessed. Therefore it was deemed safer to only remove sky pixels and avoid a scenario where an educated guess would have to be made about the colour of pixels behind a cloud. When synthetic clouds are inserted into the image, then only sky pixels are removed, not added.

Before a payload is hidden within synthetic clouds it is first compressed and encrypted. The bits of the payload are then used as input to the cloud drawing algorithm. The payload itself will determine the shape of the cloud. In theory, this method should be very resilient to statistical attacks as the edges of clouds are naturally random. When a random thing is manipulated in a random way, then there is no way of determining whether something has changed.

To increase the payload capacity of this technique, multiple natural phenomena could be combined in one image, each containing a piece of the hidden payload. A picture of a cracked road by the ocean with trees at the side of the road and clouds in the sky would contain lots of naturally random phenomena to be exploited for steganographic purposes.

All changes to the cover image are in plain sight and are visually plausible. The algorithm doesn't attempt to hide manipulations to the cover image.
Chapter 4 - Implementation

Introduction:

Initially, naturally random phenomena were compared in terms of how difficult it would be to draw them in a photo-realistic way. Clouds were a strong contender right from the start. Stars in the night sky were also considered as the arrangement of stars could be used to hide a secret message. Stars were quickly dismissed as a potential candidate for the implementation as they were deemed insecure. A steganalyst could use star maps to determine whether some stars aren’t where they should be. Grass was deemed too difficult to draw in a photo-realistic way, given the time-frame of the project.

Clouds are relatively easy to draw in a reasonably realistic way when they are far away in the distance. Attempts were made to draw large clouds however these looked very unrealistic as they were lacking 3D features. The lack of 3D features doesn’t become apparent as easily in small clouds.

The picture above shows a large cloud drawn with the latest implementation. Ignore the fact that it is unrealistically long.
Overall the large cloud looks very unrealistic. Attempts to draw realistic large clouds were given up quite early during development as the time span of the final year project was too short to modify the drawing algorithm to render 3D clouds instead of 2D clouds. The 3D cloud rendering method explained in “Modeling and Rendering Methods of Clouds” was however briefly considered until it was deemed too difficult to implement in the given time-frame.[5]

The encoding and decoding part of the cloud steganography implementation weren’t developed concurrently. First the encoding stage was developed, then the decoding stage. In hind-sight, this was a mistake. There is a trade-off between how realistic a cloud looks and how easy it is to decode. It proved to be very difficult to draw a realistic cloud and have it decode with 100% reliability. After a lot of development effort had been wasted on attempting to get both stages working reliably an intermediary “correction” stage was introduced. This stage corrects any damage caused by the blur stage that would cause the decoder to make a wrong decision. This proved to be a reliable way of ensuring 100% decoder reliability.

**Technologies used:**

- Python
- C
- OpenCV
- PAQ8PX
- Visual Studio

The vast bulk of code was written in C. Python was only used to implement a very basic GUI. The GUI was implemented using the Python Tkinter module. The purpose of the GUI is to open a Windows explorer window so that the user of the system can choose a payload file and a cover image. The python front-end then passes the file-names as command line arguments to the back-end. The application can be used without the front-end.

The compression is done by an application called PAQ8PX. According to maximumcompression.com this is the best general file compressor when measured in terms of compression ratio achieved.[6] The encryption was planned to be done within the python front-end using code taken from http://stackoverflow.com/questions/16761458/how-to-aes-encrypt-decrypt-files-using-python-pycrypto-in-an-openssl-compatible. However this stage was never integrated. Currently one must first manually compress and encrypt the payload.

The OpenCV library was used for simple image operations such as decoding compressed images, encoding uncompressed images and accessing pixels in a simple way. Furthermore it was used for conversions between colour spaces, e.g. HLS to RGB or vice versa.
Implementation stages:

The implementation can be divided into multiple stages. A brief description is provided of what each stage does in this chapter.

PAQ8PX compression:

The first stage in the application is concerned with compressing the payload. PAQ8PX is an extremely good compression software in terms of achieved compression ratio, however it is also extremely slow. The average compression speed achieved when compressing a 5.04MB JPEG was 98.4KB/s on the machine that the cloud steganography was developed on. This is very slow compared to the speeds that 7z, gzip and winrar operate at, which usually achieve compression speeds of multiple MB/s, depending on the compression settings used.

Before making a decision to use this software a test was carried out to see how well this compressor performed relative to other image compression schemes. A 20MP image of the Ha'penny Bridge in Dublin was used to test the compression schemes. In an uncompressed format, such as BMP, the image had a size of 66,431KB. When the image was converted to PNG, the resulting file size was 38,592KB. Using PAQ8PX, the achieved compressed file size was an impressive 9,197KB. After a conversion to JPEG, the file size was 3,825KB. Bear in mind though that JPEG compression results in the loss of some information whereas all other tested formats, BMP, PNG and PAQ8PX, were lossless. This test was carried out during an earlier stage of this project where the main aim was to hide an image within an image. At the time, the concept of cloud steganography hadn’t been explored yet and it was thought that the steganography method implemented would provide enough capacity to hide an image. Later, it turned out that cloud steganography offered very little payload capacity, therefore the aim shifted from hiding images within images to hiding general compressed data within images.

Using an external compression tool had the disadvantage that an extra dependency was added that made the installation process of the application more involved. Alternatively, a standard python compression module could have been used, however this wouldn’t even have come close to the compression ratios that can be achieved with PAQ8PX. As the capacity of the cloud steganography method is so limited, trading ease of installation with a higher compression ratio seemed like a good step. However, the compression must still be done manually and PAQ8PX must first be installed manually as it isn’t a part of the installation. The user doesn’t have to use PAQ8PX but it is strongly recommended.
AES encryption:

This stage takes the compressed file from the previous stage and encrypts it. This stage is an added layer of security in case the steganography is compromised. Even if an attacker is able to extract the hidden data from the stenogram (image after applying steganography), he/she would not be able decrypt the payload. This stage wasn’t implemented by the author of this report, the code to carry out the encryption was taken from (however this isn’t used in the current implementation and the payload encryption must first be carried out manually):


Crude cloud generation:

During this stage the synthetic cloud is generated. The synthetic clouds aren’t placed into the cover image automatically. The user specifies where the clouds should go by clicking twice on the image, once to define the top left-hand corner of the rectangle that the cloud should be rendered inside and once more to specify the bottom right-hand corner of the rectangle. An attempt was made to automate this process. For the application to automatically determine where clouds should be placed it would need to know what areas of the image constitute the sky, natural clouds and other things, such as the ground, trees and aircraft. A simple dynamic thresholding scheme was implemented that would partition the image into three types of areas, the sky, other clouds and everything else. The thresholding implementation was dynamic as the user would first have to click on the sky, the lowest part of the sky and a cloud. This would give the thresholding implementation a rough estimate of the colours of the sky and clouds. It would also know where the sky ended. The automation of placing clouds wasn’t developed any further than this as the thresholding proved very unreliable.

Once the user of the system has specified a rectangle to render a cloud within, the system starts drawing a cloud inside the rectangle from left to right, column by column. During the rendering process, the algorithm attempts to maintain a minimum distance to all edges of the rectangle. The starting point, i.e. the leftmost part of the cloud, is defined as halfway vertically down the rectangle and a constant offset to the right of the left border of the rectangle. Individual bits in the payload are encoded as increments/decrements of the current pixel column. This affects the width of the cloud. Being able to increment/decrement the top and bottom pixel of every column results in a capacity of 2 bits per column. The capacity could have been further increased by letting the absence of an increment/decrement mean something. This would have increased the complexity of the algorithm significantly. As this implementation is only a proof of
concept it was deemed unnecessary to increase the complexity of the system for an increase in payload capacity.

An increment represents a “1” and a decrement represents a “0”. The total capacity of a cloud is defined as:

\[ 2 \times L = C \]

L = Length of cloud in pixels  
C = Capacity measured in bits

The last two bits encoded within a cloud have a 50% chance of being wrong. This is due to the fact that on the right hand end of the cloud the top and bottom of the last column converge and overwrite each other. The decoder is aware of this issue and discards the last two decoded bits. The encoder skips back 2 bits in the payload when it moves onto the next cloud.

To prevent the cloud from taking any shape it likes the rendering algorithm follows certain rules. When a rule isn’t satisfied the system does increments and decrements on its own accord regardless of the payload. The decoder is aware of these rules and therefore knows which increments/decrements are due to rules and which increments/decrements represent bits of the payload. An example of a rule is the cloud width constraint, which forces the rendering
algorithm to maintain a minimum cloud width. This rule is only enforced in the intermediate cloud generation stage, not at the start or end, otherwise the cloud could never end. A few other rules cause the cloud to become wider and longer than the payload bits would imply.

Each pixel within the cloud is assigned a luminance value determined by a natural texture generation algorithm. The code for this was taken from http://lodev.org/cgtutor/randomnoise.html. A security requirement of cloud steganography is that the seed of the texture generation algorithm isn’t a constant value, otherwise an attacker could simply look for clouds with a particular texture.

Picture of cloud after crude cloud generation stage.
Blur:

Synthetic clouds generated by the crude cloud generation stage look very unrealistic. The blur stage essentially blurs the edges of the cloud to make the transition from sky to cloud more gradual and not as abrupt.

During this stage the application creates a sub-image (the user specified rectangle) and blurs its contents. Then the system iterates through the rectangle in the cover image column by column and replaces pixels that are edge pixels or close to the edge with the respective pixels from the blurred sub-image. Most pixels in the rectangle will remain unchanged during this process. The entire cloud rectangle isn’t blurred as otherwise the texture of the cloud would be damaged and a blur box would become visible, i.e. an area in the image where it would become apparent that a blur has been used, compromising the steganography.

Synthetic cloud after blur has been applied.
Correction:

This stage is essential as the blur stage damages the edges and may result in the decoder “taking a wrong turn” and failing to properly decode the payload. During the correction stage another pass of the rectangle is made and significant edge pixels are corrected. The vast bulk of pixels will remain untouched. Only those pixels are modified that would result in the decoder “taking a wrong turn”. After a pass has been made by this stage, one can be certain that the decoder will extract the payload with 100% accuracy.

This image highlights all edge pixels that could be modified by the correction stage in the worst case scenario, i.e. if all significant edge pixels would cause the decoder to take a wrong turn.
This is what a synthetic cloud looks like after the correction stage has been applied. Note that this is a different cloud than shown in the previous images.

**Decoding:**

The purpose of the last stage is to ensure that the extracted payload doesn't differ from the original payload. To decode a cloud steganography image a key is necessary. The key contains the payload file size and the coordinates of the fake clouds. In the current implementation the different parts of the key are stored in a linked list and displayed to the user after rendering each cloud, the key grows longer the more clouds are inserted. In a more secure and advanced implementation the key would also need to contain the seed of the cloud texture generation part, which is currently a constant value. This is a vulnerability.
Chapter 5 – Steganalysis

The field of steganalysis is concerned with attempting to expose steganography. This can be done using visual steganalysis, i.e. the human eye, or using statistical steganalysis tools.

To be fair, almost all other steganography methods are very secure when the payload is very small. Cloud steganography has a very limited payload capacity, therefore one can’t say that cloud steganography has a security advantage over other methods. Actually, the current implementation of cloud steganography is insecure. In theory though, cloud steganography should be very secure.

The above image contains one synthetic cloud with a 5 byte payload. When viewed with a normal level of zoom it is unlikely that the observer could spot the fake cloud, especially if the observer isn’t looking for one. However, once the observer zooms far into the image then the synthetic cloud may be detected.
To test the statistical resilience of cloud steganography the resulting stego image had to go through a few tests. The Simple Steganalysis Suite provided these tests. The Suite can be downloaded from:

https://code.google.com/p/simple-steganalysis-suite/

The tests carried out on the stego image were:

- LSB Enhancement
- Pixel Value
- Chi-Square
- Difference Histogram
- Primary Sets
- Neighbourhood Histogram
- Pixel Difference Histogram

All tests couldn’t detect any steganography. As mentioned earlier though, with such a small payload, most other steganography techniques would also pass these tests.
Chapter 6 – Conclusions

Overall, this project has proved that it is possible to hide data in clouds and reliably extract the payload. The implementation is a proof of concept, not a production software. The usability is poor and still has many flaws.

The invisibility (to the human eye) of the current implementation is poor, however if a more superior cloud rendering algorithm were used, the invisibility could be improved.

The payload capacity is quite low. There isn’t much that can be done about that except for combining multiple natural phenomena in one image, i.e. adding trees and waves in the water to the image. It may also be possible to somehow hide some information within the texture of the cloud, however this would probably be very difficult as natural cloud textures tend to have patterns. There may not be that much natural randomness to exploit in cloud textures.

The robustness to statistical attacks in the current implementation is decent enough. The Simple Steganalysis Suite couldn’t detect cloud steganography. However, there are some vulnerabilities that could be exploited. For example, the current implementation has a fixed seed for the random texture generation algorithm. By principle though, cloud steganography doesn’t have any known vulnerabilities. All the vulnerabilities in the current implementation could be fixed.

Possible future work around cloud steganography could attempt to render clouds using a more realistic 3D method, furthermore, the statistical vulnerabilities of the current implementation could be removed. In addition, steganography using gravel, trees or other natural phenomena could be explored and combined with cloud steganography.
Appendix – How to use

On the DVD accompanying the report is the source code of the implementation and a pre-compiled 64 bit binary for Windows machines. All required libraries are included. The binary was compiled using Visual Studio. The OpenCV vision library is a dependency of the application. The application was only tested on 64 bit Windows.

A python front-end script accompanies the application. It provides a very basic GUI. To run the application, navigate to the root directory of the provided DVD on the command line and run “python Steganography.py”. A Windows explorer window will pop up. Choose a payload file. Once a file has been chosen another Windows explorer will pop up asking for a cover image. Choose a cover image. (test.jpg in DVD root directory) Then the cover image is loaded and shown to the user. Follow these steps:

1. Click on a sky area of the image
2. Click on the lowest part of the sky, i.e. where the sky touches the ground.
3. Click on a cloud
4. Press the space key on the keyboard
5. The image is now thresholded, this may take a few seconds….
6. Now you must choose where the synthetic cloud should go. To do this imagine drawing a rectangle into which the cloud should be placed. Click on the screen where the top-left pixel of the rectangle should go.
7. Click on the screen where the bottom-right pixel of the rectangle should go.
8. Press the space key, the synthetic cloud should now be drawn, this may take a second or more.
9. If you are happy with the synthetic cloud, then go back to step 6 to draw the next synthetic cloud if the payload hasn’t been fully integrated yet. You can see how much of the payload has been integrated by checking the console window, which says how many bytes have been hidden so far.
10. If you aren’t happy with the synthetic cloud right-click anywhere on the screen before proceeding to step 6. Note that the cloud won’t disappear right away. It will disappear whilst the next cloud is being drawn.
General notes:

- The invisibility may not be very good with some images, it is known to work well with the provided image, test.jpg. This is because the base luminance value of synthetic clouds is currently hard coded and only works well with some images.
- Don’t place clouds close to edges and don’t draw extremely large clouds, no border checking is currently implemented, and therefore doing these things may result in the program not operating correctly or crashing.
- The application can’t currently be used as a standalone decoder, however after drawing a cloud the application attempts to decode the cloud and shows the resulting bits to the user. If these match the original bits (which are also displayed) then the cloud decodes fine.
- The stego image is written to the folder in which the application resides. The filename is steg.png
- Try to draw small clouds
- The application is only a proof of concept and was undergoing constant change. Therefore the usability is extremely poor, the application probably still has a lot of bugs and the source code itself isn’t optimised.
References


