Secret Communication on Facebook implemented with Browser-Based Steganography

Owen Campbell-Moore

April 7, 2013

DRAFT

Contents

1 Introduction **4**

2 Background **4**
- 2.1 Steganography . . . 4
- 2.2 Chrome Extensions and Native Client . . . 5
- 2.3 Facebook . . . 5
- 2.4 Coffeescript . . . 5
- 2.5 JPEG . . . 6
  - 2.5.1 JPEG compression and decompression . . . 6
- 2.6 Steganography in JPEG . . . 7

3 Problem Specification **8**
- 3.1 General steganography properties to be acheived . . . 8
- 3.2 Interacting with Facebook without an API key . . . 9
- 3.3 Being simple to use . . . 9
  - 3.3.1 Integrating with the browser . . . 9
- 3.4 Being subtle to use . . . 9
- 3.5 Not storing unencrypted messages or details of contacts . . . 10
- 3.6 Not changing stuck-bits of JPEGs when embedding payload . . . 10
- 3.7 Existing steganography tools are incompatible with Facebook . . . 11
  - 3.7.1 Facebook's JPEG implementation . . . 12
  - 3.7.2 Multiple JPEG compressions with the same quantisation matrix 13
  - 3.7.3 A new steganography algorithm is required . . . 14

4 Implementation **14**
- 4.1 Codes for embedding payload . . . 14
  - 4.1.1 Partitioned linear codes . . . 15
  - 4.1.2 Conceptual explanation of the code . . . 15
  - 4.1.3 Modified Linear Block Code usage . . . 16
  - 4.1.4 Code generation . . . 16
  - 4.1.5 Error correction and stuck-bit capacity of MLBCs . . . 17
  - 4.1.6 Example . . . 17
- 4.2 Coffeescript Implementation of MLBCs . . . 18
  - 4.2.1 Matrix library . . . 18
  - 4.2.2 Generating MLBCs . . . 18
  - 4.2.3 Encoding a single block using Coffeescript . . . 19
  - 4.2.4 Decoding a single block using Coffeescript . . . 20
  - 4.2.5 Encoding and decoding longer messages . . . 22
  - 4.2.6 Measuring error-correction and stuck-bit capacity of MLBCs in Coffeescript . . . 23
  - 4.2.7 Selecting the best MLBC . . . 23

4.3 JPEG encoder and decoder implementation 24
4.3.1 JPEG decoding with DCT coefficient access in Javascript . . 25
4.3.2 JPEG encoding with DCT coefficient access in Javascript . . 25
4.4 Putting the back end together 26
4.4.1 Connecting the message encoder to the JPEG encoder 26
4.4.2 Connecting the message decoder to the JPEG decoder 27
4.5 User interface implementation 28
4.5.1 Architecting the extension 28
4.5.2 Injecting code into Facebook 28
4.5.3 Hotkey based activation 29
4.5.4 Injecting a form into Facebook to create stego-objects 29

5 Analysis **30**
5.1 Error rate in practice 30
5.2 Detectability 30
5.3 User feedback and reception 30
5.3.1 Usability Studies 30

6 Conclusion **31**

7 Acknowledgements **34**

8 Appendix **34**

1 Introduction

Facebook is the most popular social network in the world with over 1 billion active users. Despite its popularity, little is offerred in the way of truly secure communication. Therefore, we present a secure mode of communication utilising Facebook as an innocent-looking medium for transmitting secret objects.

For the project, I will produce a Chrome Extension written in Javascript which allows users to embed messages up to 140 character long into any image while uploading it to Facebook. Any of the user's friends provided with the pre-shared key can then decode the hidden message using the same extension.

This application is particularly relevant at the moment since the Arab Spring of 2011 showed how wide spread use of social networking sites can be important for groups to plan uprisings and many governments are now investing heavily in systems to both access and parse messages shared on social networking websites.

The remaining sections of this project are structured as follows. In Section 2 the relevant technologies are outlined. In Section 3 the core problems I will solve are specified and in Section 4 the solutions and implementation details are presented. Experimental results are provided in Section 5 and the conclusion is drawn in Section 6.

2 Background

2.1 Steganography

Steganography is the science of hiding information. Steganography has many applications including communicating secret messages by embedding them in innocent looking cover data, often in the form of images or videos to produce a stego-object. These stego-objects are then shared on innocent channels where the payload is received by a recipient who is provided instructions on how to receive it.

The first known use of steganography was recorded in 440BC by Herodotus who describes how Demaratus, a recent King of Sparta, carved a message on the wooden surface of a tablet warning of an impending invasion of Greece before applying a beeswax surface and writing an innocent message on top. In this case the beeswax tablet provides the innocent appearing cover while only the contact who knows Demaratus' method will be able to recover the secret message.

Since the introduction of "The Prisoners Problem" by Simmons in 1983 we typically model steganography as the effort of two prisoners, Alice and Bob to communicate secretely by passing messages via a Warden. In the Passive Warden case, the Warden

is not allowed to modify the messages being passed although in the Active Warden case the Warden is allowed to tamper with or generate fake messages for Alice or Bob to attempt to decode.

2.2 Chrome Extensions and Native Client

Google Chrome is the most popular web browser in the world. It supports many progressive technologies including installable extensions which are essentially snippets of Javascript with enchanced permission to run in the background, modify sites displayed to the user, store files locally and display notifications to the user. It was selected as the target platform for this project due to its popularity and developer tools.

Native Client (NaCL) is an open-source technology which allows websites to deliver native compiled code to be run within the browser. In this way a website can deliver C code, ported to supported the security requirements of Native Client which is then run in the browser. This allows the use of existing JPEG libraries written in C within the application without sacrificing the simple browser-only user experience

2.3 Facebook

Facebook is the world's most popular social networking website. Founded in 2004, it now has over one billion active users. In August 2012 they reported an average of 300 million photos uploaded to the site every day (http://www.scribd.com/doc/103621762/Big-Data-Whiteboard-082212).

Users are able to upload an unlimited quantity of photos of up to 2048-by-2048px and view photos uploaded by their friends.

The huge amounts of innocent traffic and incredible quantity of photos being transferred makes Facebook is an ideal medium for steganography.

2.4 Coffeescript

Coffeescript is a language which compiles into Javascript. It provides Class based inheritence (which compiles into the equivalent prototypical inheritence in Javascript), provides lots of syntactic sugar and fixes much of the bizzaire beheaviour of Javascript.

Since Chrome Extensions are written in Javascript the majority of the code in this project was written in Coffeescript and then compiled into Javascript.

Here is an example Coffeescript function which cubes every value in an array (provided by the Coffeescript website):

```
cubes = (list) -> (math.cube num for num in list)
```

and its Javascript equivalent following all best practices:

```
cubes = function(list) {
  var num, _i, _len, _results;
  _results = [];
  for (_i = 0, _len = list.length; _i < _len; _i++) {
    num = list[_i];
    _results.push(math.cube(num));
  }
  return _results;
};
```

This example demonstrates the syntactic benefit of using Coffeescript over Javascript.

2.5 JPEG

JPEG is the most commonly used image format. It is used by Facebook and the majority of cameras. For that reason, my application will be based on the JPEG format for both cover images and stego-objects.

2.5.1 JPEG compression and decompression

The JPEG format is based upon the Discrete Cosine Transform (DCT), a close relative of the Discrete Fourier Transform.

The first step of JPEG compression is colour space conversion where luminance and chrominance are seperated and encoded independently. We will be hiding information only in the luminance channel of the cover image (for reasons explained in section 2.6) so the effect of compression and decompression on the luminance of an image alone is presented here. The details of chrominance compression are similar but omitted and are explained thoroughly in [x].

The image is initially divided into disjoint 8-by-8 pixel blocks which will be treated independently. Each block B undergoes the DCT to produce 64 coefficients $d_k(i)$, representing the ith coefficient of the kth block. The list of coefficients represents the weight of each mode (particular frequency) of the consine wave to be summed to reconstruct the block as illustrated in Figure 1. In this way we have separated high and low frequency components.

These coefficients are then quantised by dividing each DCT coefficient by its corresponding element from a quantisation matrix (since high frequency waves are less perceptible to humans the quantisation step divides higher frequency coefficients

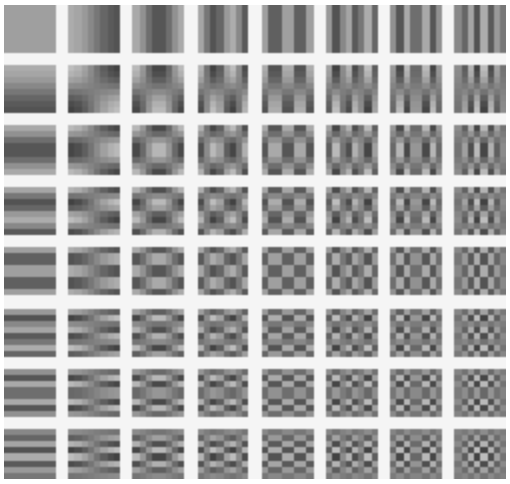

Figure 1: The 64 cosine modes of an 8-by-8 matrix

by larger values resulting in the prioritisation of lower frequency data over high) followed by a rounding to the nearest integer.

$$D_k(i) = round\left[\frac{d_k(i)}{Q(i)}\right] \tag{1}$$

This step represents a many-to-one mapping and hence is lossy. These quantised coefficients allow us to approximately reconstruct the original image by multiplying the quantised coefficient by the relevant element in the quantisation matrix and then performing the Inverse Discrete Cosine Transform (IDCT).

The quantised coefficients are finally encoded using a form of lossless encoding called Huffman coding before being written to file. The details are unimportant and therefore ommitted but explained in x.

2.6 Steganography in JPEG

Many systems have been designed to perform steganography with JPEG-compressed images forming the stego-objects. In general they operate by modifying some subset of the quantised coefficients by ± 1 to encode payload.

In almost all of them only luminance data is modified since chrominance data is compressed much more heavily so storing equivalent payload in chrominance data proves significantly more detectable.

A general embedding function is comprised of two steps: extracting the Luma DCT coefficients and applying some embedding function to modify them, encoding the

payload data.

A simple example of the process carried out by an embedding function is given:

1. Select some permutation of the extracted DCT coefficients (typically a shuffle based on a pre-shared key).
2. Perform Least Significant Bit Replacement (LSBR) on the DCT coefficients by replacing the least significant bit of each coefficient by a bit of payload.

With the corresponding extracting function:

1. Select the same permutation of the extracted DCT coefficients as used in the embedding function.
2. Map each coefficient to its Least-Significant-Bit (LSB) to obtain the original payload.

In this case a pre-shared key (of suitable length to prevent the Warden exhausing over all possible keys) determines the permutation. They key is used as a seed for pseudorandom number generater whos output is fed into an algorithm such as the Knuth Shuffle (rather than simply choosing coefficients in that order) in order to avoid choosing the same coefficient multiple times.

Many improvements on LSBR have been proposed with many contemporary systems, such as this, using F5. The F5 algorithm decrements the absolute value of a coefficient if it's LSB needs to be flipped (instead of simply replacing the least significant bit) since this turns out to be less statistically detectable.

3 Problem Specification

3.1 General steganography properties to be acheived

- Correctness: $Ext(Emb(c, k, m), k) = m$ for every cover c, key k and message m where Ext and Emb are the extracting and embedding functions respectively.
- Robustness: $Ext(Trans(Emb(c, k, m)), k) = m$ where Trans represents the transmission of the stego-object. In Passive Warden this is the identity function but in Active Warden it is considered as a function which causes some (potentially non-deterministic) errors.
- High embedding efficiency: the (average) number of payload bits hidden for each cover element changed should be as high as possible. I aim to store 140 ASCII characters of hidden message per 960-by-720px image since Twitter has demonstrated people are happy to communicate in messages of this length

and 960-by-720px is the largest size available without specifically enabling high definition images on Facebook.

- Low detectability: the warden's ability to discriminate between a stego-object and an innocent cover should be minimised as much as possible. Note that this project does not aim to acheive low detectability since it is an open problem in steganography and acheiving robustness when transmitting a JPEG with errors is the primary goal.

3.2 Interacting with Facebook without an API key

The application should not depend on Facebook's approval and should be near unblockable by Facebook.

Facebook supports a feature known as Apps where developers are provided an API key to access data for users who indicate they wish to use an app. This would provide a simple way for users to create, upload and manage their stego-objects. Unfortunately, this would leave Facebook with the power to revolk our API key so it is not an option.

Therefore a core problem to solve is how to design an application which cannot easily be disabled by Facebook but is able to interact with a users data on Facebook.

3.3 Being simple to use

A user with a techncial background should be able to send and receive messages without human assistance or prior explanation.

3.3.1 Integrating with the browser

Since the application is highly tied to Facebook I aim for all interaction with the software to happen within the browser, or if possible within Facebook itself. Hence having selected Chrome as the target browser, the main options to consider are Chrome Applications and Chrome Extensions.

3.4 Being subtle to use

Using steganography should be as subtle as possible given the nature of secret messaging. This reinforces the decision not to use a Facebook App since an App's users are listed publicly, violating users secrecy requirements. Furthermore, we should

avoid requiring users to connect to any specific server to generate stego-images since this would be easily detectable by network analysis.

A problem we therefore must solve is how to design a tool which can operate independently of the network and without disclosing its userbase publicly.

Note that we do not consider the distribution and update of the software as a problem to be solved since this is an open problem in steganography.

3.5 Not storing unencrypted messages or details of contacts

I wished to avoid the case where gaining access to a computer would allow you to easily access previous conversations or lists of contacts. The following options are therefore considered:

1. Provide contact and shared-password storage with a single master password required to access it or initiate a poll for new messages.
2. Require the user to enter their pre-shared password every time they attempt to encode or decode an image and store nothing persistantly.
3. Have the user share two keys with each of their contacts, one for revealing whether payload is stored within an image and the other for decoding the message. In this way the application could poll stored contacts and find new stego-objects, then prompting the user for the decoder password to receive the message.

Option 2 was chosen for simplicity although in future I would like to implement option 3, providing the user the option of storing a list of their contacts in the application along with the password to detect a message from said contact. The software can then allow the user to check a secret messages inbox, providing the message password to decode the messages.

3.6 Not changing stuck-bits of JPEGs when embedding payload

It is well known in the field of JPEG steganography that you should avoid modifying certain sections of an image to remain visually undetectable. A popular heuristic is to never change a coefficient set to zero. These coefficients and their corresponding least significant bit are hereby known as 'stuck'.

Since JPEG is a very efficient compressor we find on average that up to 75–95% of quantised coefficients in an image are zeros. Since we quantise higher frequency modes more greatly we find that the number of zeros, and hence stuck-bit rate increase as frequency as Figure 2 demonstrates.

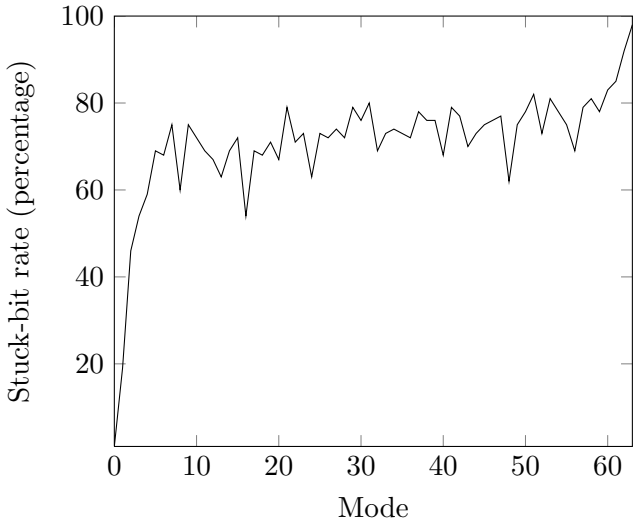

Figure 2: Average stuck-bit rates of different modes from a sample of 10 JPEGs

These high stuck-bit rates mean that steganography algorithms generally only store payload in the lowest modes, taking advantage of the lower stuck-bit rates.

In general steganography algorithms are designed to encode maximum payload whilst making the smallest number of changes by assigning many long codewords to each short message and then selecting whichever codeword best matches the coefficients. Conveniently this same idea also solves the stuck-bit problem by allowing the encoder to select an codeword which doesn't require a stuck-bit to change. (cite Jessica)

These algorithms are incompatible with an Active Warden since they require both the sender and receiver to know which bits are stuck (so they can be ignored), meaning that if any coefficients change to or from zero during recompression then decoding will fail.

Hence if Facebook's recompression introduces no errors we can implement known algorithms but if (as we shall see is the case) the JPEG recompression introduces errors then a problem we must tackle is how to design a new kind of code which can both avoid changing stuck bits and fix errors.

3.7 Existing steganography tools are incompatible with Facebook

The mathematics of JPEG compression and decompression suggests that multiple compressions with the same quality factor will not cause a change to an image's coefficients. This idea is loosly backed by the literature

8	6	6	7	6	5	8	7
7	7	9	9	8	10	12	20
13	12	11	11	12	25	18	19
15	20	29	26	31	30	29	26
28	28	32	36	46	39	32	34
44	35	28	28	40	55	41	44
48	49	52	52	52	31	39	57
61	56	50	60	46	51	52	50

Table 1: Quantisation matrix used by Facebook's JPEG implementation

> "Moreover, we also imposed that the QFs between two consecutive compression stages must differ by at least 3 units. In this way, we avoided the trivial case in which an image is recompressed with the same QF, since this would lead to ... as if no additional compression has been performed at that stage."

I hoped to exploit this by compressing images in the same way as Facebook before uploading them. They would then be recompressed with no errors and stored on the site, providing an error free channel.

The first problem with confirming this hypothesis was the lack of direct access to Facebook's JPEG implementation which was tacked by running experiments to determine the settings used by Facebook's compression algorithm, specifically the quality factor they use.

3.7.1 Facebook's JPEG implementation

The literature states that Facebook uses quality factor 85 for storing JPEGs [TODO: cite me]. To verify this, 10 uncompressed images (of dimension 960-by-720) were compressed using JPEG at Quality Factor (QF) 85 and uploaded to Facebook. Once downloaded the coefficients of the new images were extracted and compared with those of the original images. The new coefficients were found to be on average x% larger than the originals with very little variance. The test was repeated with QFs 55 and 35, resulting in average changes of y% and z% respectively.

It was observed that these values fit a straight line on a graph of QF against percentage change. Solving the equation of this line to find a change of 0% gave a QF of 75. Comparing a QF 75 JPEG's coefficients before and after re-compression by Facebook results in an average error rate of 6% (down from approximately 99% for all other QFs tested) indicating Facebook does use QF 75 when compressing images.

To confirm Facebook's use of QF 75 I used Libjpeg to extract the quantisation matrix of an image downloaded from Facebook and compared it to that of an image compressed with QF 75 by the reference JPEG implementation (provided by the Independent JPEG Group). The quantisation matrices were equal (provided in Table 1) and hence the default compression functionality of a reference implementation with QF 75 is acceptable as an approximation to that of Facebook and is as close as can be within the scope of this research. (Should I skip the above information about how I actually found QF 75 in favour of this last paragraph which was in fact the correct way to do it)

3.7.2 Multiple JPEG compressions with the same quantisation matrix

It appears that when compressing an image multiple times using the same quality factor there is unfortunately a significant amount of change to the coefficients before and after they were decompressed and recompressed.

Henceforth the coefficient-error rate is defined as the percentage of non-zero coefficients that changed after compression. We only measure the change in non-zero coefficients since as discussed earlier only these coefficients are used for storing payload. To assess the precise nature of these changes more tests were run as follows.

I selected 50 uncompressed images at random and compressed them at QF 75. I then decompressed and compressed them again at QF 75. These images exhibited a surprisingly high coefficient-error rate of 5–10%. Looking back at the JPEG compression operation I realised that images whose widths are not a multiple of 8 are padded before compression. Selecting a new sample of 50 uncompressed images with dimensions a multiple of 8 reduced this rate to 1–5%.

Note that this new requirement for a cover image's dimensions to be a multiple of 8 should be solved by the system to prevent users being required to select such images. This is another problem to solve, most likely to cropping images to the nearest multiple of 8 before embedding payload.

Experimentally I determined that the coefficient-error rate decreases by approximately 1/2 upon each decompression/recompression cycle. For example, let I_1 be a single randomly selected image initially compressed at QF 75. Let I_j be the same image decompressed and recompressed j times with QF 75. Let $\Delta(I, J)$ be the fraction of DCT coefficients which differ between I and J. When testing 50 images, we find on average:

$$\Delta(I_1, I_2) \approx 3.1\%$$
$$\Delta(I_2, I_3) \approx 1.6\%$$
$$\Delta(I_3, I_4) \approx 0.9\%$$
$$\Delta(I_4, I_5) \approx 0.7\%$$

These experimental findings were backed up by a paper published in 2010 titled 'Detecting Double JPEG Compression With the Same Quantization Matrix':

> "Furthermore, when recompressing the JPEG image over and over again, the number of different JPEG coefficients between the sequential two versions will monotonically decrease in general. For example, the number of different JPEG coefficients between the singly and doubly compressed images is generally larger than the number of different JPEG coefficients between the corresponding doubly and triply compressed images."

I suspect colour subsampling and colour space conversion are responsible for the coefficient-error rate measured since in theory the DCT and quantisation should cause no change in coefficients.

3.7.3 A new steganography algorithm is required

Existing JPEG steganography techniques all require JPEGs to be transmitted without error since in many cases a single coefficient change has the potential to completely destroy the entire payload. We have shown that this is not possible in the case of transmitting JPEGs over Facebook.

The largest problem that must therefore be solved is how to design embedding and extraction functions which can avoid modifying stuck bits and also survive the 1–5% coefficient-error rate for recompressing a QF 75 JPEG since we know that no existing steganography algorithm can do this.

Note that if we are unable to reach the robustness goal then an option would be to recompress the cover image a number of times before embedding the payload. This would exploit the above property of monatonicly decreasing coefficient-error rate but may potentially cause increased detectability.

4 Implementation

4.1 Codes for embedding payload

The first problem solved was how to design a code which can modify coefficients by ± 1 to store payload while not changing stuck bits and being resistant to an error rate of 1–5%.

It is worth recognising that pre-existing "good" steganographic codes have a very small distance between codewords so the maximum number of bits can be encoded while making a very small number of changes to the bit-stream. Conversely, error

correction codes aim to have the highest possible distance between codewords, therefore requiring a large number of changes to the bit-stream to change the payload slightly. Hence combining these two ideas is fundamentally a hard problem and there will always be a certain amount of trade-off between high capacity, low dectability properties and error-resistant properties within any code we create.

A number of potential codes were considered to solve this problem, the most promising of which were from a class known as Partitioned Linear Codes. These are presented and an alternate option will be briefly outlined in the conclusion.

4.1.1 Partitioned linear codes

Here we present a partitioned linear code first introduced in [TODO: CITE]

For this section we model the channel of transmitting images on Facebook as a noisy channel with a fixed error rate to be determined as approximatly the rate coefficients are changed non-deterministically as part of recompression.

The code presented is a Modified Linear Block Code (MLBC), capable of dealing with both random transmission errors as well as stuck bits with the assumption that the location and nature of the stuck bits are known to the encoder but not to the decoder.

4.1.2 Conceptual explanation of the code

In a partitioned linear code the encoder has a collection of error-correction codes it may use – in this way it can choose the one which most agrees with the stuck-at requirements of the transmission medium. The decoder doesn't need to know which error correction code was used due to the algebraic properties of the code as we will see so messages are decoded without the decoder ever knowing which bits were stuck.

To deal with stuck bits consider partitioning the set of all possible binary messages n-bits in length into 2^k disjoint sets $\{A_0, A_1, ..., A_{2^k}\}$ and associate a k-bit message with each subset. Now when a k-bit message $w \in \{0, 1, ..., 2^k\}$ is given to the encoder along with a description of the stuck-at bits the encoder selects a message $x \in A_w$ such that x satisfies the stuck at requirements. The decoder then identifies x as a member of A_w and can correctly decode w without knowing the location of the stuck bits.

To accommodate random errors in conjunction with stuck bits we partition error correction codes as above. The decoder receives $y = x + z$, a noisy copy of x, but x can be decoded since y was encoded with a random error correction code. The decoder may then proceed as it did previously.

4.1.3 Modified Linear Block Code usage

Let G $= [G_0^T, G_1^T]^T$ and H be the $(k+l) \times n$ generator matrix and $r \times n$ parity-check matrix where G_0, G_1 are $l \times n$, $k \times n$ matrices and $GH^T = 0$ where $n = l + k + r$. Let J be a $k \times n$ matrix such that $G_0 J = 0$ and $G_1 J = I$.

Let G_0, G_1, H and J be full rank.

To encode a $1 \times k$ message w compute $x = wG_1 + vG_0$ where v is a $1 \times l$ vector selected to maximise the agreement between x and the stuck-bits. (Note: x is now an n-bit vector and the number of stuck bits we can handle is linked to l although this relationship is unclear).

Let $y = x + z$ where z is noise (and the noise is allowed to affect the stuck bits). If $yH^T = 0$ then we expect $z = 0$ and no errors occurred since we assume the least number of errors which satisfy the equations is what actually occurred and

$$\begin{aligned} yH^T &= (x+z)H^T \\ &= (wG_1 + vG_0 + z)H^T \\ &= wG_1H^T + vG_0H^T + zH^T \\ &= w0 + v0 + zH^T \\ &= zH^T \end{aligned}$$

If $yH^T \neq 0$ then $z \neq 0$ and y contains errors which are fixed by finding the noise vector, z, with the minimum hamming weight (number of non-zero symbols), $W_h(z)$, such that $zH^T = yH^T$. Using $y = x + z$ we can now compute x.

Now the decoder has x, so compute $xJ^T = wG_1J^T + vG_0J^T = wI + v0 = w$.

4.1.4 Code generation

[TODO: CITE HEEGARD] gives a systematic form of an (n, k, l) MLBC (where n, k and 2^l are the encoded codeword length, decoded codeword length and number of encodings the encoder may choose between to avoid stuck bits respectively) with the following generators:

$$G_1 = [I_k \; 0_{k,l} \; P] \text{ and } G_0 = [R \; I_l \; Q] \tag{2}$$

Where $r = n - k - l$, P is a $k \times r$ matrix, R is an $l \times k$ matrix and Q is an $l \times r$ matrix. These give the following parity matrix and decoding matrix:

$$H = \left[-P^T \; -(Q + RP)^T \; I_r\right] \text{ and } J = [R \; I_l \; Q] \tag{3}$$

4.1.5 Error correction and stuck-bit capacity of MLBCs

For an (n, k, l) MLBC with generators G_0, G_1 and parity check matrix H define a pair of minimum distances, (d_0, d_1) such that

$$d_0 = \min_{xH^T=0,\, x\neq 0} W_h(x) \text{ and } d_1 = \min_{xG_0^T=0,\, xG_1\neq 0} W_h(x) \tag{4}$$

where $W_h(x)$ is the hamming weight of x, the number of non-zero symbols in x.

Now an MLBC with minimum distances d_0 and d_1 is t-stuck-bit, u-error correcting if and only if

$$u < \begin{cases} \frac{d_1}{2}, & \text{for } t < d_0 \\ \frac{d_1}{2} + d_0 - t - 2, & \text{for } t \geq d_0 \end{cases} \tag{5}$$

4.1.6 Example

Using systematic form we generate a (7, 2, 1) code with sending rate 2/7 capable of fixing any single error but with no guarentees on stuck bits. It has the following generator, parity and decoding matrices:

$$G_1 = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \text{ and } G_0 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{6}$$

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } J = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{7}$$

Observe that $G_0 J^T = 0$, $G_1 J^T = I$, $GH^t = 0$ and G, H and J are full rank, as required.

Now we encode $w = \begin{bmatrix} 0 & 1 \end{bmatrix}$ with the stream $s = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$ where the zero indicates that the 2nd position in the transmission medium ('stream') is stuck at 0 while the others can take either 0 or 1. Hence we take $v = \begin{bmatrix} 1 \end{bmatrix}$ so $x = wG_1 + vG_0 = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$ and the 2nd bit in the encoding now agrees with the stuck-bit in the stream.

Now let $z = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ to simulate a single error in the first position so $y = x + z = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$.

Now the decoder receives y and observes $S = yH^T = \begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix} \neq 0$ so an error is detected. The decoder proceeds by finding z such that $zH^T = S$ and $W_h(z)$ is minimised, resulting in $z = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$. Now $x = y - z$ can be calculated and $w = xJ^T = \begin{bmatrix}0 & 1\end{bmatrix}$ gives the original message despite the stuck-bit and error in transmission.

4.2 Coffeescript Implementation of MLBCs

4.2.1 Matrix library

The first task was to write a matrix library for binary matrices to be used for encoding and decoding. This was done using 2-dimentional arrays in the obvious way without more complex optimisations such as Strassen's Algorithm.

As an example, here is a snippet demonstrating adding two matrices together:

```
addMatrices = (m1, m2) ->
    if width(m1) != width(m2) || height(m1) != height(m2)
       throw "Dimentions don't match in addition"
    m = newMatrix(height(m1), width(m1))
    for row in [0..height(m1)-1]
        for column in [0..width(m1)-1]
            m[row][column] = (m1[row][column] + m2[row][column])%2
    return m
```

I also implemented functions to reduce the given matrix to row-echelon form, multiply matrices, transpose, scale by a contant, join horisontally and vertically, check for equality, calculate row rank, and calculate hamming weight. Many Javascript libraries for matrices are available but many support advanced features and use complex data-types which make adding functions or modifying beheaviour difficult. Since in total this library is less than 500 lines of simple code I deemed it small enough to be worth writing myself.

4.2.2 Generating MLBCs

Implementing MLBC generation in Coffeescript is simple, combining random matrices to genereate the systematic form as it was stated above. The only detail to be aware of is that H may not be full rank, so we introduce a loop to keep generating Hs until we find one with full rank and then proceed as expected. The Coffeescript code implementing MLBC generation is provided in Appendix 1.

4.2.3 Encoding a single block using Coffeescript

Recall that to encode a a $1 \times k$ message w we compute $x = wG_1 + vG_0$ where v is a $1 \times l$ vector selected to maximise the agreement between x and the stuck-bits.

This is implemented using a function named minimizeStuckBits which finds v and adds vG_0 to wG_1 to produce x as required. To find v such that x agrees with the stuck bits in the stream as much as possible we generate every possible vector v and attempt it, stopping early only if we find a perfect solution. This naive solution runs in $O(2^l l^2 n)$ time where l is usually no larger than 10 so this suffices. minimizeStuckBits is given below:

```
minimizeStuckBits = (G0, stream, origMessage) ->
   origStuckBitCount = width(stream) - hammingWeight(stream)
   # Generate every binary vector of length height(G0)
   vectors = allVectors(height(G0))
   minWrongBits = -1

   for vector in vectors
      matrix = [vector]
      messageToSend = addMatrices(multiplyMatrices(matrix, G0),
          origMessage)

      currentWrongBits = 0
      for column in [0..width(stream)-1]
         if stream[0][column] == 0
            if messageToSend[0][column] != 0
               currentWrongBits++

      if currentWrongBits < minWrongBits || minWrongBits == -1
         minWrongBits = currentWrongBits
         bestVector = vector
         if minWrongBits == 0
            break

   debugOutput("Best we can do is have "+minWrongBits+" stuck bits
       remaining from a starting "+origStuckBitCount)
   return addMatrices(multiplyMatrices([bestVector], G0), origMessage)
```

where allVectors was implemented in $O(n2^n)$ time as follows:

```
allVectors = (n) ->
   vectors = []
   if n == 0
      return vectors
   vectors.push [0]
   vectors.push [1]
   if n == 1
      return vectors
   for i in [0..n-2]
      count = vectors.length
```

```
11        for j in [0..count-1]
12           vectors[j+count] = vectors[j][..] #Copy the array to a new
                 location
13           vectors[j].push 0 #In the first copy add a 0 on the end
14           vectors[j+count].push 1 #In the second copy add a 1 on the
                 end
15     return vectors
```

4.2.4 Decoding a single block using Coffeescript

Decoding a message is significantly more complex. Recall that the decoder receives $y = x + z$ where z is noise and xJ^T is the original message. To calculate x we computer the syndrome $S = yH^T$ and then find the most probable noise vector, z, which minimises $W_h(z)$ such that $S = zH^T$.

Unlike encoding, the naive approach of exhausting over z quickly becomes unusable since it runs in $O(2^n nlk)$ time where n is the bit-length of the encoded message (often large). To find z I considered the problem as one of xor-satisfiability and designed a backtracking algorithm which explores solutions in order of increasing hamming weight.

Recalling that H is an $r \times n$ matrix, for each column i of the $1 \times r$ syndrome S we have

$$S_{1,i} = \bigoplus_{j=1}^{n} z_{1,j} \times H^T_{j,i} \tag{8}$$

by the definition of matrix multiplication (modulo 2). Hence we derive the following pseudo-code algorithm to construct a xor-satisfiability problem:

constraints = {}
for each column, i, of S **do**
    mustXorTo = $S_{1,i}$
    elements = {}
    for each row, j, in H^T **do**
        if $H^T_{j,i} == 1$ **then** elements = elements $\cup$ j
        end if
    end for
    constraints = constraints $\cup$ (elements, mustXorTo)
end for

We have now produced a set of constraints, each a pair $(elements_i, mustXorTo_i)$ such that

$$\bigoplus_{e \in elements_i} z_{1,e} = mustXorTo_i \tag{9}$$

The algorithm idea is to attempt to assign each variable to 0 and only re-assign it to 1 if it creates a conflict with any constraint. Initially a value MaxOnes is set to 1 and only up to that number of 1s are permitted. On each loop if no acceptable assignment can be found with the given value of MaxOnes then it is incremented. The algorithm terminates either when we've assigned the last variable and there were no conflicts or when there are no possible backtracks and we are at a conflict.

To demonstrate the algorithm in practice the following example run is provided.

Input:

$[\{elements : [0, 1, 2], mustXorTo : 0\}, \{elements : [1, 2], mustXorTo : 1\}]$

Output:

Step	Assignment	MaxOnes	Backtrack	
1	[0, undef, undef]	1	[0]	
2	[0, 0, undef]	1	[0,1]	
3	[0, 0, 0]	1	[0,1,2]	← Conflict so backtrack to 2
4	[0, 0, 1]	1	[0,1]	← Conflict so backtrack to 1
5	[0, 1, undef]	1	[0]	
6	[0, 1, 0]	1	[0]	← Conflict so backtrack to 0
7	[1, undef, undef]	1	[]	
8	[1, 0, undef]	1	[1]	
9	[1, 0, 0]	1	[1,2]	← Conflict so backtrack to 2
10	[1, 0, 1]	1	[1]	← Exceeded MaxOnes so backtrack to 1
11	[1, 1, undef]	1	[]	← Exceeded MaxOnes and no available backtracks so reset with MaxOnes = 2
12	[0, undef, undef]	2	[0]	
⋮	⋮	⋮	⋮	
21	[1, 1, undef]	2	[]	
22	[1, 1, 0]	2	[2]	← Success, return Assignment

The full code for this algorithm is given in Appendix 1 (note that many optimisations such as propagating constraints and reordering variables to try most restrictive first were implemented but made little practical difference unless $n > 50$ and since I later chose $n < 40$ they were refactored out to keep the code base as simple as possible).

Now using z, the decoder can compute $(y-z)J^T = xJ^T = wG_1J^T + vG_0J^T = wI + v0 = w$, the original message (assuming z was found correctly).

4.2.5 Encoding and decoding longer messages

The implementations provided so far can only encode a single block of binary payload. The next step was to enable encoding and decoding of long ASCII messages using an (n, k, l) MLBC. Since the medium is an stream of bits it was required for messages to include a header indicating the total message length so the decoder can unambiguously know the payload is an actual message as well as when to stop decoding.

Since I had selected the $(27, 11, 3)$ code by this point the implementation of the following assumes $k = 3$ to simplify certain aspects of padding and headers as indicated.

The first step is to convert message from ASCII into a binary vector m and pad it such that it fits neatly into blocks of length k. The end of the message was padded with $(k - |m|)\%k$ zeros (where $|m|$ is the length of the ASCII message in binary) to produce m_p.

Since $k = 3$, and a single ASCII character is 8 bytes long we can unambiguously discard any padded bits since there can never be enough padding to be mistaken for a character. If we took $k > 8$ then since 00000000 in ASCII represents the null character, the decoder will receive the original message followed by $\lfloor (k-(|m|\%k))/8 \rfloor$ null characters which was deemed also acceptable within the context of this project. Postfix Length Padding could be used in future versions if true unambiguous padding removal is required, for example if the message is in a more general encoding or $k > 8$ and ambiguous null characters are deemed unnacceptable.

Take the padded message m_p and prepend the length in blocks of the message $|m_p|/k$, followed by a terminal #, both encoded using a 3-repitition code to both ensure $|m_h|\%k = 0$ (where m_h is the padded message with header) and minimal errors occur in header transmission since this would prevent any of the message being received. The use of the 3-repition code with $k = 3$ avoided the need for header padding although in the situation with $k > 3$ we could simply pad the header after applying the 3-repition code for its error correction properties.

To decode m_h the decoder would keep accepting more input while reading only 3-encoded ASCII numbers or the terminal #. If this pattern was not found the decoder would reject its input stream and claim no message was found.

The padded message with header m_p is now left and can now be split into m_p/k blocks and encoded/decoded using the method outlined in section x, Encoding using Coffeescript. (Check: Check section correct)

4.2.6 Measuring error-correction and stuck-bit capacity of MLBCs in Coffeescript

Recall that an MLBC with minimum distances d_0 and d_1 is t-stuck-bit, u-error correcting iff

$$u < \begin{cases} \frac{d_1}{2}, & \text{for } t < d_0 \\ \frac{d_1}{2} + d_0 - t - 2, & \text{for } t \geq d_0 \end{cases} \tag{10}$$

where

$$d_0 = \min_{xH^T=0,\, x\neq 0} W_h(x) \quad \text{and} \quad d_1 = \min_{xG_0^T=0,\, xG_1\neq 0} W_h(x) \tag{11}$$

Since we are only using the lowest frequency modes we expect to have significantly fewer stuck bits to avoid than errors to correct. Therefore we may take $t = d_0 - 1$ and $u = \lfloor \frac{d_1}{2} \rfloor$.

d_0 and d_1 can be found using the same xor-satisfiability algorithm as given in section 3.3.4 with the modification such that an additional constraint function can be passed in (taking advantage of higher-order functions in Javascript) and passed the valid assignments each time one is found. In this way we can add a check at the end for d_0 that all of the variables are not assigned to zero and a check for d_1 that $xG_1 \neq 0$, in both cases causing a backtrack if they are.

TODO: Explain somewhere how we measured error rates in recompression of different modes.

4.2.7 Selecting the best MLBC

Following recompression, the average error rate for non-zero coefficients within the 1st mode of DCT coefficients was experimentally approximated as 4% in section 2. Therefore assuming all stuck-bits are avoided, a $\frac{k}{n}$ embedding efficiency and a padded message $|m_p|$ bits in length then to ensure the probability that no more than one error occurrs in message transmission is $< 95\%$ we need

Modelling the number of errors that occur when transmitting q bits as a poisson distribution with mean $\lambda = q/4$

For $n \in \{10, \ldots, 60\}$, $k \in \{5, \ldots, n-1\}$, $l \in \{0, \ldots, n-k\}$, I generated 1000 (n, k, l) MLBCs, storing a table of the number of errors u and stuck-bits t the best of the 1000 could correct for. The value of 60 as an upper bound for n was chosen to ensure the encoding and decoding times would be acceptable on average devices.

From this table a number of potential codes stood out. The best $(27, 3, 11)$ code is a 2-error, 3-stuck-bit code with over 10% message rate and seems like a good candidate. The current performance measures provide guarantees for when less than a given number of errors and stuck bits occur but say nothing about how the codes will perform in situations where error rates and stuck-bit rates are higher. Since in practice errors and stuck bits will vary with some distribution I built a simulation tool to experimentally verify whether the $(27, 3, 11)$ code could withstand real world usage. A snippet of debug output from the tool demonstrating stuck-bit avoidance and fixing errors is provided:

```
1  Now we encode w = 0,1,1
2  origMessage: 0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0
3  stream: 1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1
4  We can avoid 3 of the 3 stuck bits
5  x = wG1 + vG0 = 0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0
6  Introduced 2 errors (based on error rate of 0.04)
7  Let y = x + z = 0,1,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,0,1,1,0,0,0,0,0
8  S = yH^t = 0,0,1,1,1,1,0,1,0,0,0,1,1
9  Errors minimised by z =
       0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0 with
       hammingWeight 2
10 Attempted to fix errors, xNew =
       0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0
11 Recovered message w' = xNewJt = 0,1,1
12 0 errors occurred
```

The tool indicates that in simulation the $(27, 3, 11)$ code has an error rate of 0.01% when the underlying transmission medium has a bit error rate of 4% and stuck-bit rate of 5%. This was deemed to be suitable and I proceeded to implement the JPEG functionality of the project.

4.3 JPEG encoder and decoder implementation

Javascript is still seen as a lightweight language, uncapable in terms of performance of complex tasks such as JPEG compression and decompression, hence there is very little in terms of available code for these tasks.

Only one open source Javascript encoder and one decoder could be found online. The encoder was ported by an open source Action Script 3 by Andreas Ritter and the decoder was based on a simple open source Javascript decoder published on GitHub by notmasteryet. The decoder required only minor modifications while the encoder required major flow restructuring and bug fixing in order to allow interaction with the DCT coefficients.

4.3.1 JPEG decoding with DCT coefficient access in Javascript

Modifying the decoder was relatively simple since I could store DCT coefficients in a global variable as they became available within the flow of the code. Only minor tweaks were then required to ensure it was stored correctly.

The decoders success callback was then modified to also pass the DCT coefficients array to its calling function.

4.3.2 JPEG encoding with DCT coefficient access in Javascript

Modifying the encoder was significantly more complicated than modifying the decoder since we will need access to all the coefficients before any are written to the file in order to know how to modify them, whereas in the decoder we could simply collect the DCT coefficients and pass them out at the end.

Unfortunately the only Javascript JPEG encoder available worked by scanning vertically down the image and producing the output for each line as it went which is unsuitable for the embedding function since we need access to all of the coefficients in order to decide where to embed payload. The flow was restructured into three stages: firstly scanning the whole image to produce all DCT coefficients, then calling a passed-in function with a reference to the coefficient array, allowing it to modify them, then processing and writing the coefficients to a file.

This refactoring caused many hours of difficult bug fixing, particularly the case where one deeply call-nested array, outputfDCTQuant, an array which for each pass stores quantised DCT coefficients, was modifying references in a most unexpected way. For example, outputfDCTQuant[i] would be set to a variable x by one pass and then set to a different variable, y on the next pass. Instead of reassigning the ith cell of outputfDCTQuant, the assignment outputfDCTQuant[i] = y was evaluated as x = y, pointing x to y and causing every line of scanned coefficients to be the same as the final line. A simple outputDCTQuant = new Array(); on each pass proved to be the solution. Discovering this bug amongst such contrived and side-affect-riddled code such as **writeBits(HTAC[(nrzeroes <<4)+category[pos]]);** with variables named by scheme (e.g. tmp0p2, z3p2) marked a significant personal success within the project.

Other bugs include my choice to use i as a counter within a for-loop where some nested call also used i without the var keyword, overwriting the outer value. A final notable bug was caused by the original authors decision to name a variable fDCTQuant within a function which was also named fDCTQuant.

4.4 Putting the back end together

We now have access to an image's DCT coefficients and the ability to encode an ASCII message by providing the encoder a stream indicating which bits must remain zero and which can be changed. The next step is to piece the two together. In the case of creating a new stego-object we take a cover image and modify its DCT coefficients during compression by choosing some permutation of coefficients (based on a password) and calculating which bits are stuck, using this to inform the encoder on how to encode the ASCII message. We then modify the coefficients of the cover so the Least Significant Bits match the encoded message.

To decode the payload from a stego-object the the LSBs of the permutation of coefficients is extracted and then decoded using the technique for decoding large messages given above, correcting for errors as it goes.

4.4.1 Connecting the message encoder to the JPEG encoder

Now we have a method to gain access to the DCT coefficients of an image during encoding we need to specify how we pass information from the coefficients to the MLBC encoder along with the message and key and then use the resulting information to modify the DCT coefficients.

At a high level this process is all carried out by the following snippet:

```
1  # LUMA_ARRAY is a 2d array of blocks of coefficients belonging to
       modes. The number of blocks is given by the variable 'blocks'
2  LUMA_ARRAY = DU_DCT_ARRAY[0]
3  # List references to all the coefficients in mode 1 so we can shuffle
       them and use this order to modify them
4  coeffOrder = getValidCoeffs(LUMA_ARRAY, blocks)
5  # Apply knuth shuffle to coeffOrder
6  shuffle(coeffOrder, password)
7  # Generate the 'stream' of stuck bits from LUMA_ARRAY, accessed in the
        order determined by coeffOrder
8  stuckBitStream = coeffsToStuckBitStream(coeffOrder, LUMA_ARRAY)
9  # Apply MLBC encoding to the message using the generated stuck-bit
       stream
10 messageToHide = encodeLongMessage(mlbc, message, stuckBitStream)
11 # Modify the LUMA_ARRAY using the F4 algorithm to hide the message
12 stuckBitErrors = makeChanges(messageToHide, coeffOrder, LUMA_ARRAY)
```

The getValidCoeffs function takes D (known above as LUMA_ARRAY), an array of arrays where LUMA_ARRAY[k][i] $= D_k(i)$ represents the coefficient of the ith mode of the kth block. It returns list of coefficients which will be shuffled to create a 'coefficient order', the list representing in which order coefficients are to be modified to store payload. The indices in the coefficient order are stored in the format $64k+i$ representing the index of the ith mode's coefficient within the kth block.

The shuffle function implements an idea called Permutative Straddling to generate the order c in which to access the coefficients. This permutation is seeded by the key and used identically in both embedding and extracting so the users can store their data in a seemingly random order. This has the benefit of only allowing users with the correct key to access the message while spreading changes caused by the algorithm evenly throughout the cover. It is implemented using a Knuth Shuffle in the following way:

```
shuffle = (arr, password) ->
    # Seed the random number generator using the password
   Math.seedrandom(password)
   for i in [arr.length-1..0] by -1
      j = random(0, i)
      swap = arr[j]
      arr[j] = arr[i]
      arr[i] = swap
   Math.seedrandom()
```

The final Math.seedrandom() resets the seed for security. This is a minor concern but may prevent accidentally leaking information about the seeded password to elsewhere within the code.

The coeffsToStuckBitStream function produces an array s of 0s and 1s, where

$$s_i = \begin{cases} 1 & \text{if } D_{\lfloor (c_i/64) \rfloor}(c_i \% 64) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

where c_i is the ith element of coeffOrder and as before, $D_k(i)$ is the coefficient of the ith mode within the kth block.

The makeChanges function takes the encoded message m_h, the coefficient order c and the array of DCT coefficients D. It decrements the absolute value of $D_k(i)$ so $D_{\lfloor (c_j/64) \rfloor}(c_j \% 64) \,\%\, 2 = m_h(j)$ for $j \in \{0 \ldots length(m_h)\}$. This embedding operation is a more secure alternate to LSBR known as F5 and prevents a number of statistical steganalysis techniques. (?)

The function which carries out these steps was passed as a higher-order function into the JPEG encoder which applies it to the DCT coefficients after quantisation but before applying huffman encoding and writing them to a file.

4.4.2 Connecting the message decoder to the JPEG decoder

Connecting the decoders is similar but slightly simpler. First we generate c, the coefficient ordering and then extract the LSBs of D in that order to produce m_j, the padded message with header followed by junk, such that $m_j(i) = D_{\lfloor (c_i/64) \rfloor}(c_i \% 64) \,\%\, 2$.

The header is then decoded to allow m_p, the padded encoded message, to be extracted from m_j. m_p is then decoded to produce the original ASCII message as in section x.

4.5 User interface implementation

4.5.1 Architecting the extension

The actual encoding and decoding functions are running in a file known as a 'background script' which can communicate with other aspects of the extension, such as code injected into Facebook. A small snippet of Javascript is injected to manage decoding messages from images or opening an iframe which allows the user to create a new stego-object. Both the iframe for encoding and the decoding Javascript communicate with the background page to encode or decode messages.

This was necessary for two reasons. Firstly, because injecting complicated Javascript into Facebook itself risked code collisions and increased detectability. Secondly, since creating a new stego image happens within an iframe whose domain is the extension while decoding a message happens within the domain of Facebook so both would require an independ copy of the code due to Chrome's security features not allowing cross-domain communication. Therefore running a single copy of the encoding/decoding code and allowing sections of the extension to communicate with it was the best architectural choice.

Working with the extension messaging API in Chrome was simple but a single bug took many hours to solve, since it turned out to be a bug within Chrome itself for which I have implemented a work-around and filed a bug report on.

4.5.2 Injecting code into Facebook

Chrome Extensions allow developers to easily specify which code to inject into which website. Each extension has a manifest file where this can be specified.

In this case I added the following to my extensions manifest file:

```
"background": {
    "page": "background.html"
},
"content_scripts": [{
   "js": ["js/jquery-1.9.1.js", "js/keymaster.js", "js/inject.js"],
   "matches": ["http://www.facebook.com/*", "https://www.facebook.com
      /*"]
}],
"web_accessible_resources": [
   "index.html"
```

```
10 ]
```

which automatically launches background.html in the background and injects the three Javascript files into Facebook which which will overlay index.html on top of Facebook to provide a form for creating stego-objects.

4.5.3 Hotkey based activation

Activating the UI with hotkeys was simple. I injected a library called Keymaster, written by Thomas Fuchs, along with the following line into Facebook:

```
1 key('ctrl+alt+a', function(){ activate(); });
```

4.5.4 Injecting a form into Facebook to create stego-objects

If a user is currently viewing an image and activates the extension they will be prompted for a password and the extension will attempt to decode the stego-object. If the user is on any other page on the website they are prompted to create a new stego-object. To manage the user interface the following javascript is injected into Facebook:

```
1  function activate() {
2     var url = getImage();
3      if (url) {
4        var password = prompt("Please enter your password to decode the
             message.", "");
5        var message = decodeMessage(url, password);
6        if (message) {
7              alert("Message received:" + message);
8          } else {
9              alert("No message could be found");
10         }
11    } else { // Couldn't find the image
12       openStegoObjectCreation();
13    }
14 }
```

Where decodeMessage communicates with the background script to receive the decoded message and the openStegoObjectCreation function injects an iframe into the center of the screen, rendering the form to generate stego-objects. Injecting an iframe was preferred to adding element to Facebook since it minimises the posibility of Facebook making breaking changes or naming conflicts occurring between Facebook's and the extension's Javascript libraries.

The iframe is contained in a wrapping div in order to allow the iframe to be horisontally centered using the "margin: 0 auto" CSS property.

5 Analysis

5.1 Error rate in practice

5.2 Detectability

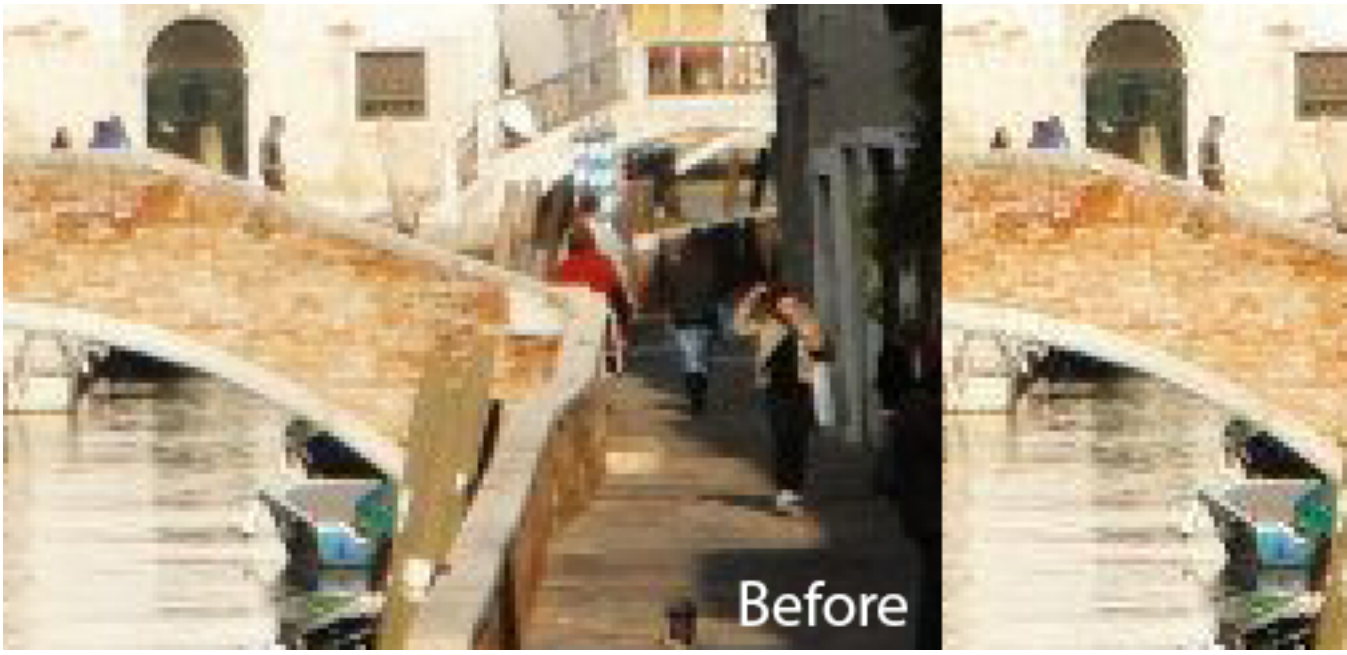

Figure 3: 200% zoom of before and after embedding a secret message. Changes are slight enough that without access to the original it would be impossible to visually detect the message.

5.3 User feedback and reception

5.3.1 Usability Studies

I carried out three usability studies on successive versions of the extension. In each case the user was asked to hide a message on a friends wall and receive a message I posted on theirs.

User 1 was a non-technical student. They were presented with a version which required them to chose an image whose dimensions are a multiple of 8. This proved too difficult and the user became frustrated. The user also failed to rename the stego-object something ending in .jpg when prompted so they were unable to reupload it

to facebook. These issues were solved by using an HTML5 canvas to automatically crop the cover and by a new Chrome feature which allows developers to prompt file downloads to bear a certain name.

User 2 was a computer scientist. They remarked "hey, it's pretty easy to use" and successfully created a stego-object without any prompts. They then proceeded to upload the cover image instead of the stego-object but realised when the message wouldn't decode. Although they blamed themselves I added a further reminder to the instructions regarding this. They commented "the decoder works great" and successfully received the message posted on their wall.

User 3 was a non-technical student. They remarked the instructions were too long so a 'Quick start' section was introduced. They successfully created a stego-object and posted it on Facebook without prompts. They also successfully received a message, remaking 'it's very easy'.

The usability is therefore determined to be suitable, especially considering the application is mostly of interest to technical individuals.

6 Conclusion

A Chrome Extension has been presented enabling Facebook users to communicate secretly and securely. This is the first publicly available steganographic system for JPEGs which supports recompression as found on almost all popular websites.

Alternate codes for embedding (redraft? Shorten? Lengthen?)

I selected Modified Linear Block Codes for their stuck-bit-avoidance and error-correction properties. An alternate option to avoid stuck bits would be to modify Least Significant Bits in the same way as before such that 00 encodes a zero and 01, 10 or 11 encode a 1.

The only case where encoding fails is encoding a 1 where the two adjascent bits on the stream are both stuck at 0. An error correction code would be nested within this to correct for these errors. Assuming a stuck-bit rate p, error rate q and a code word with even 0s and 1s this situation only occurs in $\frac{p^2}{2}$, giving a new error rate q' of $\frac{p^2+2q(q-1)p(1-p)+q^2(1-p)^2}{2} + \frac{2q(1-q)+q^2}{2}$ (I can justify this, not sure if it's a good use of space?). When plotted we see that if $p < 0.2$ then $q' < q$ and q' grows almost linearly to p so we can remove the stuck-bit rate on the channel in exchange for halving sending rate and a linear increase in error rate in terms of the stuck-bit rate.

Hence nesting a strong error correction code such as a Turbocode within this code to reduce the stuck-bit rate could provide better results than obtained above using

MLBCs and critically would increase capacity by allowing the use of other modes with higher stuck-bit rates.

Reflecting on User Interface design

Given the nature of the application great consideration was given to the user interface and interaction in order to allow maximum discression and security. During the course of building the application the user interface changed significantly from being push-based with contact storate, polling and notifications to a discrete hotkey-activated pull system where no past messages or data on the user's contacts are stored.

Initially I decided the system should be a Chrome Application with a launch icon on the New Tab Page allowing the user to manage their contacts and stored passwords as well as send new messages. The first version polled Facebook every minute and scanned contacts new uploads for hidden messages, popping up a notification with the message if one was found. I later deemed this undesirable since it would be easy for somebody other than the user with access to the computer to quickly collect the passwords and list of contacts.

I also decided that subtlety was likely to be a priority for users so this was added to the problem specification and the application was designed around a hotkey-based activation system where users press ctrl+alt+a while on Facebook to reveal the user interface to send or receive messages without notifications or otherwise visible UI.

These changes resulted in throwing away large amounts of code for polling Facebook and persistent secure storage (non-trivial in only a web browser without a server connection) so in the future I will spend more time deciding on user interfaces up front to avoid wasting effort.

Reflecting on technology choice

Initially I selected Google's Native Client technology for the JPEG and coefficient modification functionality since it allows standard C libraries such as Libjpeg to be used. I wrote embedding and extracting functions in C before recompiling them using a toolchain provided by Google to allow the code to be run by Chrome securely.

Code compiled to Native Client is severly limited and sandboxed to prevent mallicious attacks. This applies particularly to memory management and passing data between the tab process and the Native Client process. These restrictions proved insurmountable, with the task of passing a local JPEG into the specially compiled Libjpeg library taking several days due to outdated documentation and lack of debugging tools. For this reason I decided to move the entire project to Javascript and rewrite the JPEG embedding and extraction procedures.

Achieving capacity goals

The extension acheives a capacity of approximately 136 ASCII characters per 960-by-720px image, almost reaching my goal of 140 characters. The fundamental limitation on capacity when using Modified Linear BLock Codes is which modes of DCT coefficients can be used. In this application only mode 1 (of 0–63) were used since MLBCs can only deal with relatively low (realistically up to 20% stuck-bit rate) and the higher frequency modes all incur much higher stuck-bit rates ranging from ∼45% for mode 2 to ∼98% for mode 63. Other codes would be required if these modes are to be used for capacity in the future.

The requirement of using only a single mode results in high statistical detectability since acheiving full capacity of 120 characters requires the utilisation of every coefficient in the mode, changing on average 50% of them which causes significant statistical (although not visible) change.

Achieving robustness

Write me after analysis is completed

Applications and Ethics

It would be irresponsible to release any new security tool without discussing its applications and ethics, especially given the inherent dangers of an easy-to-use secret communication channel.

> Can steganography be used as a terrorist tool? The answer is an unequivocal yes. The possibility for the abuse of steganography techniques by terrorists is obvious. ... Whether these techniques have been used yet is still open to debate, but the opportunity for terrorists to add steganography to their tool kit is undeniably at hand.

The extension built for this project is not deemed a large threat since the robustness requirements meant we must make a very large number of changes to the image. This was acceptable since undetectability was never a goal for the project. The extension is therefore of interest to hobbyists and researchers to a far greater extent than it is to terrorists.

7 Acknowledgements

8 Appendix

```
# Generate an MLBC in systematic form
newMLBC = (n, k, l) ->
    debugOutput("This is an ("+n+", "+k+", "+l+") code with sending
        rate " + k/n)

    r = n - k - l

    if (2^r) > n
        debugOutput("Increase n or decrease k or decrease l. You can't
            make H full rank with this.")
        return {success: false}

    correct = false
    attempts = 0
    # Keep trying to generate an MLBC until H rank == n
    while !correct

        # Generate random matrices here.
        P = newRandomMatrix(k, r)
        Q = newRandomMatrix(l, r)
        R = newRandomMatrix(l, k)

        # Generate H according to spec
        Pt = transpose(P)
        RP = multiplyMatrices(R, P)
        QplusRPt = transpose (addMatrices(Q, RP))
        Ir = identity(r)
        H = horisontalJoin(horisontalJoin(Pt, QplusRPt), Ir)

        correct = (columnRank(H) == n)

        attempts++
        if count > 100000
            debugOutput("We tried 100000 matrices. Giving up.")
            return {success: false}

    # Generate G1 according to spec
    Ik = identity(k)
    zeroskl = newMatrix(k, l)
    G1 = horisontalJoin(horisontalJoin(Ik, zeroskl),P)

    # Generate G0 according to spec
    Il = identity(l)
    G0 = horisontalJoin(horisontalJoin(R, Il), Q)

    # Generate J according to spec
```

```
    Ik = identity(k)
    Rt = transpose(R)
    zeroskr = newMatrix(k, r)
    J = horisontalJoin(horisontalJoin(Ik, Rt), zeroskr)
    debugOutput("J:")
    debugOutput(matrixString(J))

    # Generate G by joining G0, G1
    G = verticalJoin(G0, G1)

    # Generate more useful things for later use
    Ht = transpose(H)
    Jt = transpose(J)

    # Verify the checks work:
    if !verifyMLBCCorrectness(G, G0, G1, Ht, Jt, n, k, l, r)
        throw "MLBC is invalid"

    return {k: k, n: n, l: l, r: r, G1: G1, Ht: Ht, Jt: Jt, G0: G0}
```

```
# Constraints is an array of objects like {elements: [0,1,2], mustXorTo: 1}
solveMinimally = (variableCount, constraints) ->

   # Takes a single constraint and an assignment and returns whether the constraint is violated
   satisfies = (constraint, assignment) ->
      xor = 0
      for element in constraint.elements
         # If one of the variables hasn't been assigned
         if assignment[element] == undefined
            # Since it could be either 0 or 1 the constraint isn't violated
            return true
         xor = (xor+assignment[element])%2
      return xor == constraint.mustXorTo

   # assignment[i] represents the assignment to element i. Initially undefined forall i
   assignment = []
   # We progress left to right while assigning. This is a list of indices we can backtrack to and restart from there
   validBacktracks = []
   # These are loop counters for timeout
   looped = 0
   loopLimit = 10000000
   # We explore the options in order of increasing hamming weight by allowing up to maxOnes ones and incrementing this value
   maxOnes = 1
   # How many ones are currently in the assignment?
   currentOnes = 0
   # Which element are we currently assigning?
```

```
27    current = 0
28
29    # While we haven't assigned every variable
30    while current < variableCount
31       looped++
32       break if looped > loopLimit
33
34       debugOutput "Current: "+current+" assignment: "+assignment.
             toString()+" validBacktracks: "+validBacktracks.toString()+"
              currentOnes: "+currentOnes+" maxOnes: "+maxOnes
35
36       # Always try 0 first
37       if assignment[current] == undefined
38          assignment[current] = 0
39          # This is backtrackable since we could change it to a 1
40          validBacktracks.push current
41       # We've backtracked so try setting this to a 1
42       else if assignment[current] == 0
43          assignment[current] = 1
44          currentOnes++
45       # assignment[current] == 1 will never happen since current would
              not have been in validBacktracks
46
47       backtrack = false
48       # Check whether we've violated any constraints and potentially
             backtrack
49       for constraint in constraints
50          if !satisfies(constraint, assignment)
51             backtrack = true
52             break
53       # If we've assigned more ones than allowed
54       if currentOnes > maxOnes
55          backtrack = true
56
57       if backtrack
58          debugOutput "We assigned "+current+" to "+assignment[current
                ]+" and it violated so backtrack to the last pos in ["+
                validBacktracks.toString()+"]"
59          if validBacktracks.length > 0
60             current = validBacktracks.pop()
61             # Decrement currentOnes for every 1 in assignment after
                   current
62             if current+1 <= variableCount-1
63                for i in [current+1..variableCount-1]
64                   if assignment[i] == 1
65                      currentOnes--
66             # Reset assignments for variables with index > current
67             assignment.length = current + 1
68          else # No backtrack options, can we increase maxOnes?
69             if maxOnes < variableCount
70                debugOutput "Nowhere to backtrack to so increment
                      maxOnes and go back to the start"
```

```
71                maxOnes++
72                # Reset the whole assignment and restart
73                assignment.length = 0
74                current = 0
75                currentOnes = 0
76             else
77                debugOutput Nowhere to backtrack to and maxOnes is
                      already "+variableCount+" so no assignment is valid
                      "
78                return {success: false}
79          else # If we didn't have to backtrack, simply continue!
80             debutOutput "We assigned "+current+" to "+assignment[current
                   ]+" and didn't violate so continue."
81             current++ #We didn't have to backtrack and we're not done,
                   increment and loop!
82
83       # Since we're out of the loop either we assigned the last variable
             and there was no conflict or we timed out
84
85       if looped > loopLimit
86          debugOutput "Gave up!"
87          return {success: false}
88       debugOutput "Loop finished because we assigned the last variable
             and it didn't violate. We took "+looped+" loops"
89       return assignment
```