An Error-Resistant Steganography Algorithm For Communicating Secretly On Facebook

Owen Campbell-Moore

Abstract

This project provides a method for performing steganography on Facebook by hiding messages in photos using a browser extension. Much effort was expended ensuring messages survive JPEG recompression given the added restriction that certain 0s within the image were not to be modified, a combination not previously studied. The goal was to build a highly usable system which provides a payload of approximately one “tweet” (140 characters) per 960-by-720px image without becoming detectable to the naked eye. The extension is now available on the Chrome Web Store and has been downloaded over 8000 times.

Department of Computer Science, Oxford University

Contents

Chapter 1

Introduction

Facebook is the most popular social network in the world with over 1 billion active users [1]. Despite its popularity, little is offered in the way of secure or secret communication. Therefore, I present a secure mode of communication which utilises photo sharing on Facebook as an innocent-looking medium for transmitting secret messages.

The aim of this project is to produce a Chrome Extension written in Javascript which allows users to secretly embed up to 140 characters of data into any image which is then transmitted using Facebook. Any of the user's friends provided with the pre-shared password can then decode the secret message from the image using the same extension, despite the image having been recompressed by Facebook.

Communication via social-networking sites contributed to the 2011 Arab Spring in which many people overthrew their local regimes [2]. The provision of secure communication tools built upon social media is now vital to protect free speech online since many governments are investing in systems to access and parse private messages shared on such websites.

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

Chapter 2

Background

2.1 Steganography

Steganography is the art of concealed communication where the very existence of the message is secret [3]. Innocent 'cover' data, often in the form of images or videos, are modified very slightly to produce a 'stego-object' containing the secret message. These stego-objects are then shared on innocent channels from which the payload is received by a recipient provided with the decoding instructions.

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's method will be able to recover the secret message [4].

Since the introduction of 'The Prisoners' Problem' in [5] we typically model steganography as the effort of two prisoners, Alice and Bob, to communicate secretly by passing messages via a Warden who observes all communication.

2.1.1 Active and Passive Wardens

When the Warden is allowed to tamper with the messages being passed, we refer to them as an *Active Warden.*[1] If tampering is disallowed, we refer to them as a *Passive Warden.*

An 'intentional' Active Warden attempts to destroy any hidden message while best retaining the appearance of the image. In contrast, an 'unintentional' Active Warden may make incidental modifications which damage hidden payload but this was not a goal for them. For example, a noisy channel is an unintentional Active Warden.

The channel formed by transmitting messages via JPEGs which will be recompressed is therefore recognised as an unintentional Active Warden since errors will occur but

[1]The Active Warden is also sometimes permitted to generate fake messages for Alice or Bob to attempt to decode although we shall not consider that case here.

are not intended to destroy payload (although they almost always do).

The majority of research into JPEG Steganography has assumed Passive Wardens (i.e. images transmitted without modifications). Since our system exhibits an Active Warden, much of the work presented is original.

2.2 Chrome Extensions and Native Client

Google Chrome is the most popular web browser in the world [6]. It supports many progressive technologies including installable extensions which are essentially snippets of Javascript with enhanced 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) [7] 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, specially compiled to meet 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]

2.3 Facebook

Facebook is the world's most popular social networking website. Founded in 2004, it now has over one billion active users [1]. In August 2012 they reported an average of 300 million photos being uploaded to the site every day [8].

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

The very large amount of innocent traffic and quantity of photos being transferred makes Facebook an ideal medium for steganography.

2.4 JPEG

JPEG is the most commonly used image format on the internet [9]. 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]My original intention was to write the embedding and JPEG functions in C and run them using Native Client. For technical reasons, I eventually decided to write the whole project in Javascript. A discussion of this decision can be found in Section 6.2.

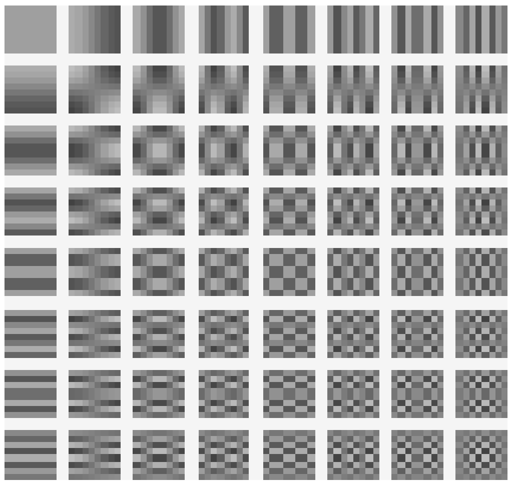


Figure 2.1: The 64 cosine modes of an 8-by-8 matrix

2.4.1 JPEG compression and decompression

The JPEG format is based upon the Discrete Cosine Transform (DCT), a close relative of the Discrete Fourier Transform. Relevant portions of JPEG compression are given here and further details are explained thoroughly in [10].

The first step of JPEG compression is colour space conversion where luminance (brightness) and chrominance (colour) data are separated and encoded independently. We will be hiding information only in the luminance channel of the cover image (for reasons explained in Section 2.5) so the effect of compression and decompression on the luminance of an image alone is presented here.

The image is initially divided into disjoint 8-by-8 pixel blocks which will be treated independently. Each block undergoes the DCT to produce 64 coefficients, $d_k(i)$, representing the ith coefficient of the kth block. We number these coefficients 0–63. The transform function is given below for completeness although its details need not concern us.

$$d_k(u+8v) = \sum_{x=0}^{7}\sum_{y=0}^{7} \alpha(u)\beta(v) g_{x,y} \cos\left[\frac{\pi}{8}(x+\frac{1}{2}u)\right] \cos\left[\frac{\pi}{8}(y+\frac{1}{2}v)\right] \tag{2.1}$$

where u is the horisontal spatial frequency, for the integers $0 \leq u < 8$,

v is the vertical spatial frequency, for the integers $0 \leq v < 8$,

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{8}}, & \text{if } u = 0 \\ \sqrt{\frac{2}{8}}, & \text{otherwise} \end{cases}$$

$g_{x,y}$ is the pixel value at coordinates (x, y) within the kth block,

$d_k(u+8v)$ is the $u+8v$th DCT coefficient of the kth block.

The list of coefficients for each block, d_k, represents the weight of each mode (particular frequency) of the cosine wave to be summed to approximately reconstruct the block as illustrated in Figure 2.1. 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 by larger values, resulting in the prioritisation of lower frequency data over high) followed by a rounding to the nearest integer.

$$D_k(i) = round\left[\frac{d_k(i)}{Q(i)}\right] \tag{2.2}$$

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. Details can be found in [10] but are unimportant to this project.

2.5 Steganography in JPEG

Many existing systems perform steganography with JPEG-compressed images as stego-objects. In general, they operate by modifying some subset of the quantised coefficients by ± 1 to encode payload [12], creating an image which cannot be distinguished from the original by the human eye.

The Human Visual System is much more sensitive to brightness than colour so chrominance data is stored with much lower resolution than luminance data. Embedding equivalent quantities of payload in chrominance, as opposed to luminance, is therefore significantly more detectable. For this reason, only luminance data are changed by the majority of steganography systems.

A general embedding function consists of two steps: extracting the Luma DCT coefficients and then applying some embedding function to modify them to represent the payload data [13].

F5 [22] is a commonly implemented embedding operation which modifies the least significant bits (LSBs) of the DCT coefficients by decrementing their absolute value (as opposed to simply replacing the LSBs of the cover with payload since F5 turns out to be significantly less detectable). It is defined here and implemented in the project.[3]

[3] The actual algorithm given here is F3. For historic reasons this is known as F5 since they are equivalent with the exception that F5 avoids a security flaw in F3 which our application is not susceptible to.

Consider some permutated subset, $\{e_0, \dots, e_n\}$, of the extracted DCT coefficients and a message $M = \{m_0, \dots, e_n\}$.

Algorithm 1 F5's embedding function

```
for each $m_i$, of $M$ do
    if $LSB(e_i) \neq m_i$ then
        if $e_i > 0$ then
            $e_i \leftarrow e_i - 1$
        else
            $e_i \leftarrow e_i + 1$
        end if
    end if
end for
```

Algorithm 2 F5's extraction function

```
for each $e_i$, in $\{e_0, \dots, e_n\}$ do
    $m_i = LSB(e_i)$
end for
```

In this example, the permutation of DCT coefficients may be derived from a pre-shared password so only the sender and receiver know which order to read/write the bits. Modifying and reading the coefficients in such an order is known as *permutative straddling*. It ensures the changes are distributed evenly throughout the image while providing additional security since the correct password is required to determine the order in which bits should be read [22].

2.5.1 Indirect representations

Contemporary steganography tools often utilise an indirect representation of payload in order to achieve a higher embedding rate (bits of payload stored per bit of cover changed) compared to simply embedding the raw payload in the cover [22].

With each possible binary message, m, we associate a set of possible encodings, $\{c\}$ (with $|c| > |m|$)[4]. The association is defined by a multivalued encoding function $E : \{0,1\}^{|m|} \to \mathcal{P}\left(\{0,1\}^{|c|}\right)$ with a corresponding decoding function, D, defined such that $c \in E(m) \implies D(c) = m$.

E is designed such that for every possible message, m, and initial cover sequence, s, there exists an encoding $c \in E(m)$ such that the hamming distance between s and c is minimal. The encoder can then select the representation with the least hamming distance from what already exists in the cover, allowing it to convey significantly higher payload per bit changed than is otherwise achievable [16].

[4]Note that no two distinct messages share any encodings, i.e. $E(m) \cap E(m') = \emptyset$ for every $m \neq m'$.

2.6 Coffeescript

Coffeescript is a language which compiles into Javascript. It provides syntactic sugar, fixes much of the bizarre behaviour of Javascript and supports class-based inheritance (which compiles into prototypical inheritance in 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 [11] which cubes every value in an array:

```
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. Refer to [11] for more details on Coffeescript.

2.7 Distinction between steganography and watermarking

Watermarking is the process of embedding a robust marker into an image such that the marker can survive heavy operations, such as cropping and resizing, being applied to the image. They are primarily used to identify the copyright holder of images but have many other uses.

In contrast with steganography, watermarking is primarily concerned with preventing the removal (intentional or otherwise) of the marker while Active Warden steganography is primarily concerned with the undetectability of payload with robustness as a secondary concern.

Watermarking techniques are therefore allowed to ignore heuristics used to minimise detectability which must be observed by steganography.

Chapter 3

Problem Specification

3.1 General steganography properties to be achieved

- Correctness: $Ext(Emb(c, p, m), p) = m$ for every cover c, password p and message m where Ext and Emb are the extracting and embedding functions respectively.
- Robustness: $Ext(Trans(Emb(c, p, m)), p) = m$ where $Trans$ represents the transmission of the stego-object. In Passive Warden this is the identity function but in Active Warden it is 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 payload 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. *Note that this project does not aim to achieve low detectability since it is an open problem in steganography. Achieving robustness while transmitting JPEGs with errors is the primary goal.*

3.2 Interacting with Facebook without an API key

There is some chance that Facebook would not approve of being used as a medium for steganography. The application should therefore not depend on Facebook's approval in terms of an API key or otherwise.

Facebook supports a feature called Apps where developers are provided an API key to access the data of users who indicate they wish to use the 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 revoke 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 user's data on Facebook.

3.3 Providing discretion for the user

Given the nature of secret messaging, steganography tools should preferably be subtle to use. This reinforces the decision not to use a Facebook App since an App's users are listed publicly. 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.

3.4 Being simple to use

Any user with a technical background should be able to send and receive messages without human assistance or prior explanation of the application.

3.4.1 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 and ruled out Facebook Apps, the main options remaining are Chrome Applications and Chrome Extensions.

3.5 Not storing unencrypted messages or contact details

I wished to avoid the case where gaining access to a computer would allow you to easily access previous secret conversations or a list 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 persistently.
3. Require the user to share two passwords 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.

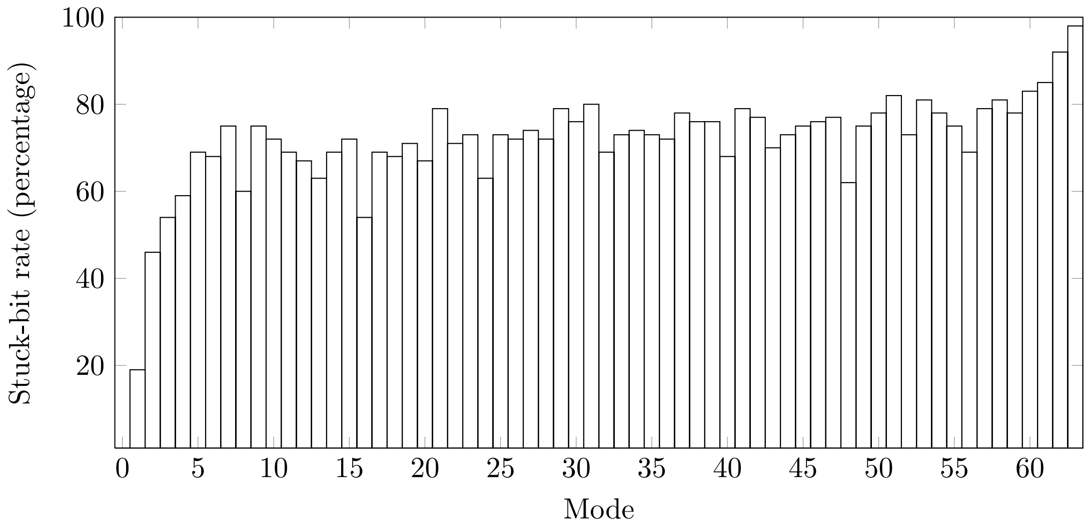


Figure 3.1: Average stuck-bit rates within modes from a sample of 10 JPEGs

Option 2 was chosen for simplicity, although in the 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 could then allow the user to check a “secret messages inbox”, where they would provide the password to decode the message contents.

3.6 Not changing stuck-bits of JPEGs

It is well documented in the steganography literature that embedding algorithms should preserve certain features of an image to remain undetectable. Every non-trivial steganography algorithm avoids changing coefficients from zero to any other number since this operation is known to be highly detectable. This fact is so well studied that we shall assume it as a requirement and not explore it further [16].

These zero coefficients and their corresponding least significant bit which must not be changed 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 apply greater quantisation to higher frequency modes, the number of zeros, and hence stuck-bit rate, increases as frequency increases. This is demonstrated in Figure 3.1.

Note that mode 0, the ‘DC mode’, has negligible stuck-bit rate. The DC mode is encoded differently from the remaining 63 modes in a way such that hiding data in it is highly detectable. For this reason, it is ignored by steganography algorithms [15].

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 [18].

Wet Paper Codes [16] are most frequently used for avoiding stuck-bits, but are incompatible with an Active Warden since they require both the sender and receiver to know

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 3.1: Quantisation matrix used by Facebook's JPEG implementation

which bits are stuck, meaning that if any coefficients change to or from zero during recompression then decoding will fail.

Hence, if Facebook's recompression introduces no errors then we can implement a known encoding and embedding method 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 also fix errors.

3.7 Existing steganography tools incompatibility with Facebook

The mathematics of JPEG compression and decompression suggest that multiple compressions with the same quality factor (QF) will not cause a change to an image's coefficients. This idea is loosly backed by the decision in [19] to omit studying the effect of compressing an image multiple times with the same quality factor to avoid "the trivial case".

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 QF they use.

3.7.1 Facebook's JPEG implementation

The literature states that Facebook uses QF 85 for storing JPEGs [20]. An attempt was made to verify this by uploading an image compressed at QF 85 to Facebook and regarding the change in coefficients. The coefficients changed significantly more than expected and futher experimentation indicated that Facebook now uses QF 75 to store images.

To confirm this finding 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 a reference JPEG implementation (provided by the Independent JPEG Group).

The quantisation matrices were equal (provided in Table 3.1), 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 project.

3.7.2 Compressing an image multiple times with the same quantisation matrix

Since Facebook compresses every uploaded image with QF 75, it is of critical importance to understand the effect of decompressing and recompressing a JPEG with the same quantisation matrix. Experimentally, it was established that compressing an image multiple times with the same QF causes a non-trivial amount of change to the coefficients before and after. To assess the precise nature of these changes, more tests were run as follows.

I selected 50 uncompressed images at random from the IAN Image library and compressed them at QF 75. I then decompressed and recompressed them at QF 75. These images exhibited a surprisingly high coefficient-error[1] rate of 5–10%. It is a known phenomenon that errors occur upon compressing images whose widths are not a multiple of 8 since they are padded before compression. Selecting a new sample of 50 uncompressed images whose dimensions are a multiple of 8 reduced this rate to 1–5%.[2]

I experimentally determined that the coefficient-error rate decreases by approximately 1/2 upon each decompression/recompression cycle (later verified by [21]). 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, I found on average:

$$\begin{aligned}
&\Delta(I_1, I_2) \approx 3.1\% \\
&\Delta(I_2, I_3) \approx 1.6\% \\
&\Delta(I_3, I_4) \approx 0.9\% \\
&\Delta(I_4, I_5) \approx 0.7\%
\end{aligned}$$

Note that unlike stuck-bit rates, non-zero error rates do not vary dramatically between modes. I will therefore select an encoding method capable of correcting error rates of up to 5% while maximising the stuck-bit rate it can deal with. I will then select the maximum number of modes possible to store payload subject to the condition that the average stuck-bit rate of the selected modes is less than the stuck-bit rate our encoding method can handle.

[1]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.

[2]This new requirement for a cover image's dimensions to be a multiple of 8 should be solved by the application to prevent users being required to select such images. The solution will most likely be to crop images to the nearest multiple of 8 before embedding payload.

Why this is not the identity operation

It was observed above that taking a compressed image, decompressing it and compressing it again at the same quality factor was not the identity operation as I initially supposed.

This is most likely due to the DCT and IDCT rounding floating point values and also producing values which fall outside the valid range [0..255] altogether. Such values must be clamped to either 0 or 255 [14].

Note that colour subsampling and colour space conversion may also play some role in the errors experienced.

3.7.3 A new steganography algorithm is required

A review of the literature indicates that existing JPEG steganography techniques (which avoid stuck-bits) require JPEGs to be transmitted without error if stuck-bits are to be avoided, since in many cases a single coefficient change has the potential to destroy the entire payload. I have also shown that it is impossible to avoid errors when uploading JPEGs to Facebook since multiple compressions always cause small changes, even in the case where QF match across compressions.

The largest problem that must therefore be tackled is to design embedding and extraction functions which can avoid modifying stuck bits and also survive the 1–5% coefficient-error rate for doubly-compressing a QF 75 JPEG.

Note that we could sacrifice security by decompressing and recompressing the cover multiple times before embedding payload. This would decrease the coefficient-error rate in exchange for introducing a signature. We will therefore only exploit this property if we are unable to reach the robustness goal set for the project.

Chapter 4

Implementation

4.1 Robust encodings which avoid modifying stuck-bits

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%.

By applying the technique of indirect representations presented in Section 2.5.1, we are able to avoid modifying stuck bits by first filtering possible representations to only those which would not modify any stuck bits and then selecting the one with the least hamming distance from the cover.

Selecting the encodings carefully allows redundancy to be included in the codewords which can be used for error correction. Unfortunately this idea of combining error-correction with a code to avoid stuck bits is harder than it appears as I will now explain.

Representing payload indirectly requires codewords for distinct messages to have very small hamming distances between one another so minor changes can vastly affect payload. This is at odds with error correction which aims to maximise the 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 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.

I considered a number of potential codes to solve this problem. The most promising were from a class known as Partitioned Linear Codes which are presented and implemented below.

4.1.1 Modified Linear Block Codes

The code presented is a Modified Linear Block Code (MLBC) from the class of Partitioned Linear Codes 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 [17].

4.1.2 Conceptual explanation of the code

In a partitioned linear code, the encoder has a collection of error-correction codes from which to choose – in this way it is able to choose the one which most agrees with the stuck-at requirements of the transmission medium. The decoder does not 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 $w \in \left\{0, 1, ..., 2^k\right\}$ 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 $G_1 J = I$.[1]

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.

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 $xJ^T = wG_1J^T + vG_0J^T = wI + v0 = w$.

[1]Where capitals represent matrices and lowercase characters represent either vectors or dimensions.

4.1.4 Code generation

The systematic form of an (n, k, l) MLBC is given in [17] (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:

$$G_1 = [I_k \; 0_{k,l} \; P] \text{ and } G_0 = [R \; I_l \; Q] \tag{4.1}$$

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

$$H = \begin{bmatrix} -P^T & -(Q + RP)^T & I_r \end{bmatrix} \text{ and } J = [R \; I_l \; Q] \tag{4.2}$$

To generate any MLBC we can therefore simply choose random P, Q and R matrices and then use the above forms.

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

$$d_0 = \min_{\substack{xH^T=0 \\ x\neq 0}} W_h(x) \text{ and } d_1 = \min_{\substack{xG_0^T=0 \\ xG_1\neq 0}} W_h(x) \tag{4.3}$$

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 [17]

$$u < \begin{cases} \frac{d_1}{2}, & \text{for } t < d_0 \\ \frac{d_1}{2} + d_0 - t - 2, & \text{for } t \geq d_0 \end{cases} \tag{4.4}$$

4.1.6 Example

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

$$G_1 = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \text{ and } G_0 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{4.5}$$

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } J = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.6}$$

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

Note that the all the theory, algorithms and implementation details that follow are original work unless otherwise stated.

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-dimensional 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 "Dimensions 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 constant, 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 behaviour difficult. Since writing my own appeared easy I deemed it small enough to be worth writing myself (it turned out to be 300 lines of simple code).

4.2.2 Generating MLBCs

Implementing MLBC generation in Coffeescript is simple, combining random matrices to generate 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 the appendix.

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.

For this, I designed a function named FindBestX which finds the best v produces $x = vG_0 + wG_1$ 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.

Algorithm 3 FindBestX

```
function FindBestX($wG_1$, $G_0$, $stream$, $l$)
    $x \leftarrow$ undefined
    for each $v$ in $\{0,1\}^l$ do
        $x' \leftarrow wG_1 + vG_0$
        $x'$.stuckBitsMissed $\leftarrow 0$
        for each $stream_i$ in $stream$ do
            if $stream_i = 0$ and $x'_i \neq stream_i$ then
                $x'$.stuckBitsMissed++
            end if
        end for
        if $x$ is undefined or $x'$.stuckBitsMissed $<$ $x$.stuckBitsMissed then
            $x \leftarrow x'$
        end if
    end for
    return $x$
end function
```

where my implementation which generates $\{0,1\}^l$ runs in $O(l2^l)$ time and is given:

```
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
    for j in [0..count-1]
      vectors[j+count] = vectors[j][..] #Copy the array to a new
        location
      vectors[j].push 0 #In the first copy add a 0 on the end
      vectors[j+count].push 1 #In the second copy add a 1 on the end
  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 compute 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.[2]

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

$$S_{1,i} = \bigoplus_{j=1}^{n} z_{1,j} \cdot H^T_{j,i} \tag{4.7}$$

by the definition of matrix multiplication (modulo 2). Hence I derived the following pseudo-code algorithm to construct an XOR-satisfiability problem:

Algorithm 4 Constructing an 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 ∪ $j$
        end if
    end for
    constraints ← constraints ∪ (elements, mustXorTo)
end for
```

We have now produced a set of constraints, $\{(elements_i, mustXorTo_i)\}$ such that

[2]I referred to the literature for efficient XOR-satisfiability solvers. These exist but are complex and designed for problems much larger than those which occur within this application. For that reason I devised my own simpler backtracking algorithm.

$$\bigoplus_{e \in elements_i} z_{1,e} = mustXorTo_i \tag{4.8}$$

The algorithm idea for solving such a problem 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 the appendix (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).

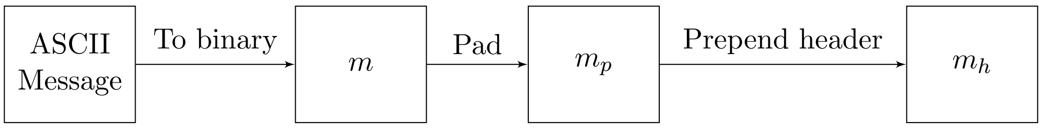


Figure 4.1: The steps involved in encoding long messages

4.2.5 Transmitting longer messages

The implementation 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 a stream, 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.[3]

A number of methods exist to efficiently construct headers which can be unambiguously removed. For this application, I assume the message and header are subject to the following conditions:

1. The header must convey the length of the message
2. The header must be unambiguously removable
3. The header must be padded (or otherwise modified) to fit in blocks of size k
4. The header should occupy minimal space
5. The whole message (including header) should be padded (or otherwise modified) to fit in blocks of size k

Encoding with k = 3

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

First convert the message from ASCII into a binary vector m and pad it with $|m| \bmod k$ zeros to produce m_p. This padding ensures that it fits neatly into blocks of length $k = 3$.[4]

A header is required to ensure the correct number of blocks are decoded (and indeed to recognise that a message is hidden). The terminal # is appended to the padded message's length[5], $|m_p|/k$, forming the header, h. The header is then encoded with a 3-repetition code to prevent errors from damaging it and also to ensure it fits correctly into blocks of size $k = 3$ without padding.

[3]Note that including a recognisable header may present a security flaw. The alternative is to force all messages to take a constant length would require the algorithm to modify coefficients needlessly for shorter messages. Since statistical undetectability wasn't a primary goal the header method was selected.

[4]Since $k = 3$ and a single ASCII character is 8 bytes long, the decoder can unambiguously discard any padded bits since there can never be enough padding to be mistaken for a character.

[5]The length is in blocks and represented in ASCII for simplicity

We now compute the message with header $m_h = h||m_p$ where $||$ represents sequence concatenation. This is the final message and is ready to be encoded.

Decoding with k = 3

The decoder takes $m_j = m_h||z$ where $z \in \{0,1\}*$, the message with a header, followed by the remainder of the stream, as input. It should keep accepting input while reading 3-repetition encoded ASCII numbers or the terminal #. If this pattern was not found the decoder should reject its input stream and claim no message was found.

Using the length found in the header, the decoder can now take the correct amount of input, unambiguously remove the header to produce m_p and then decode each block individually as described in Section 4.2.4.

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 if and only if [17]

$$u < \begin{cases} \frac{d_1}{2}, & \text{for } t < d_0 \\ \frac{d_1}{2} + d_0 - t - 2, & \text{for } t \geq d_0 \end{cases} \tag{4.9}$$

where

$$d_0 = \min_{\substack{xH^T=0 \\ x\neq 0}} W_h(x) \quad \text{and} \quad d_1 = \min_{\substack{xG_0^T=0 \\ xG_1\neq 0}} W_h(x) \tag{4.10}$$

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 $u = \lfloor \frac{d_1}{2} \rfloor$.

We can find d_0 and d_1 using the same XOR-satisfiability algorithm as given in Section 4.2.4 with the modification that an additional constraint function can be passed in (taking advantage of higher-order functions in Javascript) which checks 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.

4.2.7 Selecting the best MLBC

For $n \in \{10, \dots, 60\}$, $k \in \{5, \dots, n-1\}$, $l \in \{0, \dots, n-k\}$, 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 given 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:

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

In simulation the $(27, 3, 11)$ code has an error rate of 0.001% when the underlying transmission medium has a bit error rate of 4% and stuck-bit rate of 20%.

960-by-720px images have 10800 8-by-8px blocks and hence 10800 sets of coefficients per mode which can be modified. With a 3/27 embedding rate this gives an effective capacity of 1200 bits (150 characters) per mode. Since using only mode 1 provides capacity in excess of my goal and has stuck-bit rate $\approx$ 20% I proceeded to implement the JPEG functionality of the project, aiming to use only mode 1 for storing payload.

4.3 JPEG encoder and decoder implementation

Javascript is still seen as a lightweight language, incapable of performaning complex tasks such as JPEG compression and decompression, hence there is very little code publicly available for these tasks.

Only one open source JPEG encoder and decoder written in Javascript could be found online. The encoder [24] was ported from an open source Action Script 3 encoder and the decoder [26] was published on GitHub. 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 decoder's success callback was then modified to also pass the DCT coefficients array to the continuation provided by 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. This approach 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 the whole image is scanned to produce all the DCT coefficients; secondly they are passed to a higher order function supplied to the encoder which modifies them. Finally, the encoder is allowed to finish processing the coefficients and writes the data to a file.

This refactoring caused many hours of difficult bug fixing, particularly in one case where a deeply call-nested array, outputfDCTQuant, which stores quantised DCT coefficients during each pass, 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 author's decision to name a variable fDCTQuant within a function which was also named fDCTQuant.

4.4 Connecting the JPEG implementation to the message encoder/decoder

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 are stuck and 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), calculating which bits are stuck and then using this to inform the encoder on how to encode the ASCII message. We then modify the coefficients of the cover using the F5 operation (outlined in Section 2.5) so the least significant bits of the stego-object match the encoded message.

To decode the payload from a stego-object the LSBs of the permutated subset of coefficients are extracted and the technique outlined above for decoding large messages is applied, correcting for errors as it goes.

4.4.1 Connecting the message encoder to the JPEG encoder

Now that 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 password and then use the resulting information to modify the DCT coefficients.

This is all carried out by a function passed into the encoder and is illustrated by the following snippet:

```
# LUMA_ARRAY is a 2d array of blocks of coefficients belonging to modes.
     The number of blocks is given by the variable ‘blocks’
LUMA_ARRAY = DU_DCT_ARRAY[0]
# List references to all the coefficients in mode 1 so we can shuffle
    them and use this order to modify them
coeffOrder = getValidCoeffs(LUMA_ARRAY, blocks)
# Apply knuth shuffle to coeffOrder
shuffle(coeffOrder, password)
# Generate the ‘stream’ of stuck bits from LUMA_ARRAY, accessed in the
    order determined by coeffOrder
stuckBitStream = coeffsToStuckBitStream(coeffOrder, LUMA_ARRAY)
# Apply MLBC encoding to the message using the generated stuck-bit
    stream
messageToHide = encodeLongMessage(mlbc, message, stuckBitStream)
# Modify the LUMA_ARRAY using the F5 algorithm to hide the message
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 a 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 the 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 generates the order c in which to access the coefficients. This permutation is generated by a Knuth Shuffle which utilises a pseudorandom generator seeded by the password. This permutation is 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 password to access the message while spreading changes caused by the algorithm evenly throughout the cover (which minimises detectability and is a technique known as Permutative Straddling [22]). The implementation is given:

```
shuffle = (arr, 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()
```

Note that the final Math.seedrandom() resets the seed for enhanced 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

$$s_i = \begin{cases} 0 & \text{if } D_{\lfloor (c_i/64) \rfloor}(c_i \bmod 64) = 0 \\ 1 & \text{otherwise} \end{cases} \tag{4.11}$$

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. Intuitively $s_i = 1$ means that the ith bit can be changed to 0 if necessary while $s_i = 0$ indicates that the ith bit is stuck and must remain equal to 0.

The makeChanges function implements the F5 embedding operation outlined in Section 2.5. It 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 $D_{\lfloor c_j/64 \rfloor}(c_j \bmod 64) \bmod 2 = m_h(j)$ for $j \in \{0 \ldots \mathit{length}(m_h)\}$.

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 order, as above and then extract the LSBs of D in that order to produce m_j, the padded message with header followed by junk, such that $m_j(i) = D_{\lfloor c_i/64 \rfloor}(c_i \bmod 64) \bmod 2$ where $m_j(i)$ is the ith value in the binary vector m_j.

m_j is then decoded to produce the original ASCII message as in Section 4.2.5.

4.5 User interface implementation

4.5.1 Connecting the extension's components

The actual encoding and decoding functions are running in a 'background script' within Chrome which can communicate with other aspects of the extension, such as code injected into Facebook. A small snippet of Javascript is injected into Facebook to manage decoding messages from images and 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 independent 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 relatively simple but a single bug (where a message would simply go missing) took many hours to solve, since it turned out to be a bug within Chrome itself. I implemented a temporary workaround and the bug has now been fixed in the current version.

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 extension's manifest file:

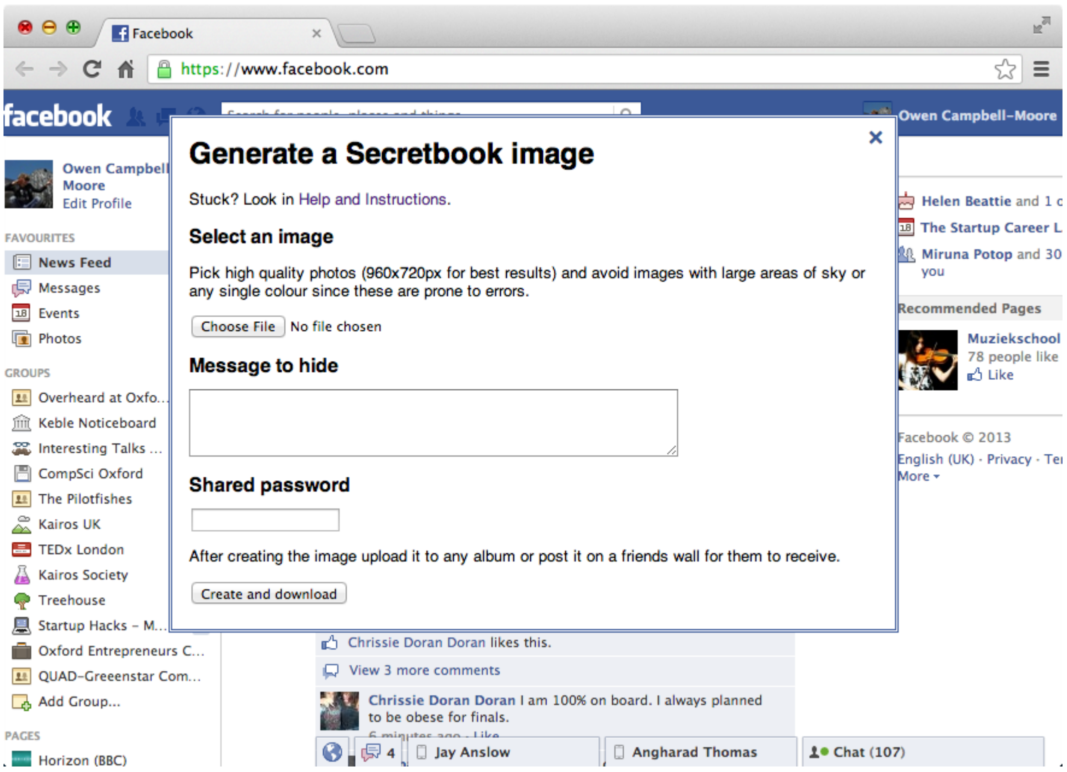
```
"background": {
    "page": "background.html"
},
"content_scripts": [{
   "js": ["js/jquery-1.9.1.js", "js/keymaster.js", "js/inject.js"],
   "matches": ["http://www.facebook.com/*", "https://www.facebook.com
       /*"]
}],
"web_accessible_resources": [
   "index.html"
]
```

which automatically launches background.html in the background and injects the three Javascript files into Facebook 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 the Keymaster library [25], along with the following line into Facebook:

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



Generate a Secretbook image

Stuck? Look in Help and Instructions.

Select an image

Pick high quality photos (960x720px for best results) and avoid images with large areas of sky or any single colour since these are prone to errors.

Choose File No file chosen

Message to hide

Shared password

After creating the image upload it to any album or post it on a friends wall for them to receive.

Create and download

Chrissie Doran Doran I am 100% on board. I always planned to be obese for finals.

Figure 4.2: Screenshot of the user interface injected into Facebook for creating stego-objects

4.5.4 Injecting a form into Facebook to create stego-objects

If the user is not currently looking at an image on Facebook when they press the hotkey they are prompted to create a new stego-object. To manage the user interface the following javascript is injected into Facebook:

```
function activate() {
   var url = getImage();
    if (url) {
      var password = prompt("Please enter your password to decode the
          message.", "");
      var message = decodeMessage(url, password);
      if (message) {
            alert("Message received:" + message);
        } else {
            alert("No message could be found");
        }
   } else { // Couldn't find the image
      openStegoObjectCreation();
   }
}
```

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 possibility 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.

If a user is currently viewing an image when they press the hotkey then they will be prompted for a password and the extension will attempt to decode the stego-object.

Chapter 5

Evaluation

5.1 User feedback and reception

I wrote a blog post on April 7th announcing the release of the extension and submitted it to Hacker News. The post trended to the top of the home page, resulting in its discovery by tech journalists.

Wired and Mashable both requested interviews and published articles documenting the extension. These lead to the announcement's discovery by the Daily Mail who published a factually incorrect article including the subtitles "First time anyone has worked out how to hide messages in computer files" and "There is a fear that the technology could be used by terrorists". This article was then discovered by CNN and NBC News, the former running a TV segment on the extension and the latter publishing an article to their website.

A search on Google News now returns over 120 articles in various languages documenting the extension. These articles have now been tweeted over 5k times and according to the Chrome Web Store, the extension has 8,240 users [23].

This reception was far beyond anything I had expected and discounting the Daily Mail article, I was very satisfied with the response.

5.1.1 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 friend's '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 choose an image whose dimensions are a multiple of 8. This proved too difficult and the user became frustrated. The user also failed to append the download with .jpg when prompted and were hence unable to re-upload it to Facebook. These issues were solved by using an HTML5 canvas to automatically resize and crop the cover and by using a trick which allows developers to prompt data URI downloads to bear a certain name (given in the appendix).

User 2 was a computer scientist working with the improved version. 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’ chapter 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.

5.2 Error rate in practice

To assess error rate in practice, 200 large JPEG photos were selected from the IAN Image Library as a representative sample to be tested. I hid a 100-character random message into each and then uploaded them to Facebook. They were then downloaded and the encoded payload was compared to the decoded payload.

122 errors occurred out of a total 168000 bits transmitted, giving a 0.000726 bit-error rate. This can also be viewed as a 0.0058 character-error rate or a 0.609 message-error rate (one character or more incorrect).

Hence average messages experience minor errors frequently but messages are by-and-large able to be understood.

5.3 Detectability

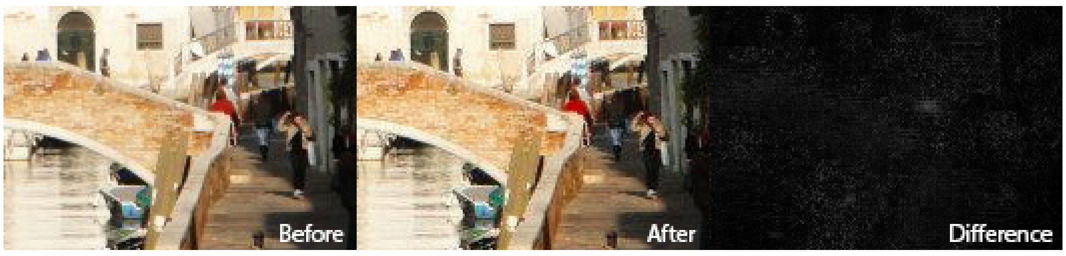


Figure 5.1: Before and after embedding a secret message and a difference with high contrast all viewed at a zoom of 200%. Changes are slight enough that without access to the original it would be impossible to visually detect the presence of a secret message.

By simulating filling capacity it was established that hiding random 136-character messages in 200 large JPEG photos resulted 855631/1679467 $\approx$ 49% of the non-zero coefficients of mode 1 being altered.

On average, each image had 446126 non-zero coefficients (in total across modes 1–63), so this represents a modification of 0.97% of the non-zero coefficients available in all modes.

Unsurprisingly, strong correlations exist between the various modes within an image. For that reason, changing (on average) 50% of DCT coefficients within an individual mode is trivial to detect. This is a mildly disappointing result since we only modify 0.97% of coefficients overall but by clustering the changes in mode 1, the application causes itself to be easily detectable.

5.4 Average encoding and decoding time

Hiding random 100-character messages in the same 200 large JPEG photos took 268 seconds on a Macbook Air with a 1.8GHz dual-core Intel Core i5. This is equivalent to an average encode time of 1.3 seconds. This is deemed acceptable since the process of creating a stego-object is relatively infrequent and waiting for one to two seconds is not a surprising or uncomfortable delay for such a process.

Decoding the same messages took 116 seconds, giving an average decode time of 581ms per image. Displaying the message half a second after being provided a decoding password is fast enough that users do not wonder if an error has occurred. If this delay increased significantly then some form of activity notification would be desirable, but in this case it isn't deemed necessary.

Chapter 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 avoids modifying stuck bits while supporting recompression as found on most social networking sites.

This is distinct from low-detectability watermarking algorithms since we have successfully preserved ‘stuck-bits’ within a robust framework, where watermarking would have overwritten them.

6.1 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 discretion and security. During the course of building the application the user interface changed significantly from being push-based with contact storage, polling and notifications to a discrete hotkey-activated pull system where no past messages or data on the user’s contacts are stored.

I initially decided the project 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 local storage (non-trivial in only a web browser without a server connection). In the future I will spend more time deciding on user interfaces up front to avoid wasting effort.

6.2 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 limited and sandboxed to prevent malicious 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.

Despite its reputation as a slow language, Javascript proved suitably fast for this application.

6.3 Achieving capacity goals

The extension achieves a capacity of 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 achieving full capacity of 140 characters requires the utilisation of every coefficient in the mode, changing on average 50% of them which causes significant statistical (although not visible) change.

6.4 Achieving robustness goals

The application exhibits a character-error rate of 0.0058. This proves suitable for transmitting short messages due to the high redundancy of English. If binary data were to be transmitted then another error-correction code would have to be nested within the current one, reducing capacity further, or a new form of code would have to be used which can provide higher error-correction capabilities as well as stuck-bit avoidance.

6.5 Achieving visual undetectability

The resulting images are successful in passing visual inspection. Filling payload capacity causes us to change on average, 50% of the mode 1 coefficients which results in high statistical detectability. This is unfortunate but acceptable within the requirements of this project.

6.6 Future work

This application has effectively converted the problem of private communication to one of key exchange. It remains for secure, secret key exchange to be solved in steganography as it was in cryptography by techniques such as Diffie-Hellman and Public Key Cryptography.

In the future I would like to include support for symmetric encryption to prevent attackers guiding a brute force by exploiting predictable patterns caused by MLBC encoding. This would reduce capacity but protect messages from being read even if they are detected.

I would also like to try other approaches to encoding such as nesting powerful error correction codes within more naive stuck-bit avoiding codes.

Since the majority of image hosting websites compress images similarly to Facebook, it would also be nice to build a more general version of the extension by parameterising the embedding and extraction functions to take a quality factor which could be extracted from images hosted on the site.

Acknowledgements

I would like to thank Dr Andrew Ker for suggesting such a challenging and interesting project and for imparting some of his excellent knowledge of steganography. His attentive assistance was invaluable to this project.

I would also like to thank Andreas Ritter [24] and notmasteryet [26] for their Javascript-based JPEG Encoder and Decoder without which I would most likely still be wrestling with a Native Client version of the project.

Bibliography

[1] Facebook, 2013. *Facebook Reports First Quarter 2013 Results.* Accessible from: http://investor.fb.com/releasedetail.cfm?ReleaseID=761090 [Accessed 3 May 2013].

[2] Howard, P.N., et al., 2011. *Opening Closed Regimes: What Was the Role of Social Media During the Arab Spring?.* Project on Information Technology & Political Islam.

[3] Fridrich, J, 2009. *Steganography in Digital Media: Principles, Algorithms, and Applications.* Cambridge University Press, Cambridge.

[4] Herodotus, 1996. *The Histories.* Penguin Books, London. Translated by Abrey de Sélincourt.

[5] Simmons, G. J, 1983. *Prisoners' problem and the subliminal channel.* In: Advances in Cryptology: Proceedings of CRYPTO 83. D. Chaum, ed. Plenum, pp. 51-67.

[6] Cullen, A., 2012. *Independent web analytics firm StatCounter confirms milestone as Chrome overtakes IE globally for first calendar month.* Accessible from: http://gs.statcounter.com/press/chrome-overtakes-ie-globally-monthly [Accessed 17 April 2013].

[7] Yee, B., et al., 2009. *Native Client: A Sandbox for Portable, Untrusted x86 Native Code.* 30th IEEE Symposium on Security and Privacy, pp. 79-93.

[8] Facebook, 2012. *Big Data Whiteboard.* Accessible from: http://www.scribd.com/doc/103621762/Big-Data-Whiteboard-082212 [Accessed 17 April 2013].

[9] Morkel, T., et al., 2005. *An Overview Of Image Steganography.* Proceedings of the Fifth Annual Information Security South Africa Conference (ISSA2005), Sandton, South Africa, June/July 2005.

[10] Hung, A.C., 1993. *PVRG-MPEG CODEC 1.1.* Department of Computer Science, Stanford University.

[11] Coffeescript. *Coffeescript homepage and examples.* Accessible from: http://coffeescript.org/ [Accessed 16 April 2013].

[12] Fridrich, J., et al., 2007. *Statistically undetectable jpeg steganography: dead ends challenges, and opportunities.* MM&Sec 2007, pp. 3-14

[13] Johnson, N.F. and Katzenbeisser, S., 2000. *A Survey of steganographic techniques.* S. Katzenbeisser and F. Petitcolas (Eds.): Information Hiding, pp. 45-49.

[14] Newman, R., 2002. *A Steganographic Embedding Undetectable by JPEG Compatibility Steganalysis.* Proc. Information Hiding Workshop, pp. 261.

[15] Sallee, P., 2005. *Model-based methods for steganography and steganalysis.* International Journal of Image and Graphics 5(1), pp. 167190.

[16] Fridrich, J., et al., 2004. *Writing on wet paper.* In: ACM Workshop on Multimedia and security, Magdeburg, Germany.

[17] C. Heegard, 1983. *Partitioned linear block codes for computer memory with stuck-at defects.* IEEE Trans. Inf. Theory, vol. IT-29, no. 6, pp. 831842.

[18] Goel, M.K. and Jain, N., 2008. *A Novel Visual Cryptographic Steganography Technique.* International Journal of Computer, Electronics & Electrical Engineering Volume 2 Issue 2 pp. 40.

[19] Milani, S., Tagliasacchi, M., Tubaro, S., 2012. *Discriminating multiple JPEG compression using rst digit features.* In: Proc. of the IEEE ICASSP, pp. 3.

[20] Castiglione, A. et al., 2011. *A forensic analysis of images on online social networks.* Third International Conference on Intelligent Networking and Collaborative Systems, pp. 682.

[21] Huang, F. et al., 2010. *Detecting double JPEG compression with the same quantization matrix.* IEEE Transactions on Information Forensics and Security, 5(4):848-856.

[22] Westfeld A., 2001. *F5A Steganographic Algorithm: High Capacity Despite Better Steganalysis.* Proc. 4th Internationall Workshop Information Hiding, Springer-Verlag, pp. 289302.

[23] Chrome Web Store. *Secretbook.* Accessible from: http://goo.gl/o7Pth [Accessed 18 April 2013].

[24] Ritter, A., 2009. *Javascript JPEG encoding.* Accessible from: http://ajaxian.com/archives/javascript-jpeg-encoding [Accessed 18 April 2013]

[25] Thomas Fuchs, 2012. *Keymaster.* Accessible from: https://github.com/madrobby/keymaster [Accessed 20 Dec 2012]

[26] Notmasteryet, 2011. *Simple JPEG/DCT data decoder in JavaScript.* Accessible from: https://github.com/notmasteryet/jpgjs [Accessed: 18 April 2013]

Appendix

Generating 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}
```

Algorithm for solving XOR-satisfiability problems

```
# 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 = []
   # 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?
   current = 0

   # While we haven't assigned every variable
   while current < variableCount

```

```
      debugOutput "Current: "+current+" assignment: "+assignment.toString()+" validBacktracks: "+validBacktracks.toString()+" currentOnes: "+currentOnes+" maxOnes: "+maxOnes

      # Always try 0 first
      if assignment[current] == undefined
         assignment[current] = 0
         # This is backtrackable since we could change it to a 1
         validBacktracks.push current
      # We've backtracked so try setting this to a 1
      else if assignment[current] == 0
         assignment[current] = 1
         currentOnes++
      # assignment[current] == 1 will never happen since current would not have been in validBacktracks

      backtrack = false
      # Check whether we've violated any constraints and potentially backtrack
      for constraint in constraints
         if !satisfies(constraint, assignment)
            backtrack = true
            break
      # If we've assigned more ones than allowed
      if currentOnes > maxOnes
         backtrack = true

      if backtrack
         debugOutput "We assigned "+current+" to "+assignment[current]+" and it violated so backtrack to the last pos in ["+validBacktracks.toString()+"]"
         if validBacktracks.length > 0
            current = validBacktracks.pop()
            # Decrement currentOnes for every 1 in assignment after current
            if current+1 <= variableCount-1
               for i in [current+1..variableCount-1]
                  if assignment[i] == 1
                     currentOnes--
            # Reset assignments for variables with index > current
            assignment.length = current + 1
         else # No backtrack options, can we increase maxOnes?
            if maxOnes < variableCount
               debugOutput "Nowhere to backtrack to so increment maxOnes and go back to the start"
               maxOnes++
               # Reset the whole assignment and restart
               assignment.length = 0
               current = 0
               currentOnes = 0
            else
               debugOutput Nowhere to backtrack to and maxOnes is already "+variableCount+" so no assignment is valid"
               return {success: false}
      else # If we didn't have to backtrack, simply continue!
         debutOutput "We assigned "+current+" to "+assignment[current]+" and didn't violate so continue."
         current++ #We didn't have to backtrack and we're not done, increment and loop!
```

```

  # Since we’re out of the loop either we assigned the last variable
      and there was no conflict or we timed out

  debugOutput "Loop finished because we assigned the last variable and
      it didn’t violate.
  return {success: true, assignment: assignment}
```

Prompting a file download with a specific name

This method exploits the new ‘download’ attribute provided by Chrome which allows data URI links to suggest a name for the downloaded file. I create a link with such an attribute and then fire a click event using Javascript as follows:

```
function downloadWithName(uri, name) {
  var link = document.createElement("a");
  link.download = name;
  link.href = uri;
  link.fireEvent('onclick');
}
```