TEXT BASED STEGANOGRAPHY USING GOLAY CODES

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Aniket Ashok Dixit
Summer 2010
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Aniket Ashok Dixit:

Text Based Steganography Using Golay Codes

J. Carmelo Interlando, Chair
Department of Mathematics and Statistics

Joseph Lewis
Department of Computer Science

William Root
Department of Computer Science

07.15.2010
Approval Date
ABSTRACT OF THE THESIS

Text Based Steganography Using Golay Codes
by
Aniket Ashok Dixit
Master of Science in Computer Science
San Diego State University, 2010

Steganography is the art and science of transmitting hidden messages. In modern communications systems, this means hiding information in communication media such as audio, text, and images. Ideally, except for the sender and receiver, no third party should even suspect the existence of such messages. Digital communications systems require the use of error-correcting codes (ECC) to combat noise, or errors, introduced by the corresponding (communication) channel. Basically, an ECC adds redundancy to a message so that the errors introduced by the channel can be corrected.

In our context, the code redundancy can be utilized to insert stega bits (that is, bits of a secret message) masked in the form of artificial errors, which in turn, cannot be distinguished from genuine channel errors. Therefore, noisy communication channels provide a suitable framework for steganography. In this work, we focus on text-based steganography. The underlying ECC is the Golay code, which breaks down the information sequence into blocks of 12 bits. At the end of the encoding process, each 12-bit block is transformed into a 23-bit block, called a codeword. The Golay code is capable of correcting up to three errors in a block of 23 bits and is attractive for combating errors in very noisy communication channels. Two modes of insertion of stega bits are discussed and compared. The modes represent a trade-off between accuracy and secrecy. In the first, a more accurate version of the secret message is recovered in comparison with the second; however, it is more susceptible to being detected by an eavesdropper than the second mode.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ix</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 REQUIREMENTS</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Platform Requirements</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Functional Requirements</td>
<td>4</td>
</tr>
<tr>
<td>3 TECHNOLOGY</td>
<td>6</td>
</tr>
<tr>
<td>3.1 Java</td>
<td>6</td>
</tr>
<tr>
<td>3.1.1 Simplicity</td>
<td>7</td>
</tr>
<tr>
<td>3.1.2 Robustness</td>
<td>7</td>
</tr>
<tr>
<td>3.1.3 Multithreading</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Java AWT Components</td>
<td>7</td>
</tr>
<tr>
<td>3.3 Subversion Repository</td>
<td>8</td>
</tr>
<tr>
<td>4 NOISY CHANNELS AND ERROR CORRECTING CODES</td>
<td>11</td>
</tr>
<tr>
<td>4.1 Noise</td>
<td>11</td>
</tr>
<tr>
<td>4.2 Binary Symmetric Channel</td>
<td>12</td>
</tr>
<tr>
<td>4.3 Linear Codes</td>
<td>13</td>
</tr>
<tr>
<td>4.4 Golay Codes and Applications</td>
<td>14</td>
</tr>
<tr>
<td>5 ENCODING OF STEGANOGRAPHIC MESSAGES</td>
<td>17</td>
</tr>
<tr>
<td>5.1 Flow of Encoding Mechanism</td>
<td>17</td>
</tr>
<tr>
<td>5.2 Repetition Codes</td>
<td>18</td>
</tr>
<tr>
<td>5.3 Golay Codes Encoding</td>
<td>20</td>
</tr>
<tr>
<td>5.4 Insertion of Stega Bits into the Cover Media</td>
<td>26</td>
</tr>
<tr>
<td>5.4.1 Mode 1</td>
<td>26</td>
</tr>
</tbody>
</table>
5.4.2 Mode 2 ..................................................................................................................27

6 DECODING OF STEGANOGRAPHIC MESSAGES ................................................30
   6.1 Parity Check Matrix..........................................................................................30
   6.2 Mode 1 Decoding............................................................................................32
   6.3 Mode 2 Decoding with Detection Rule .........................................................34

7 OBSERVATIONS AND CONCLUSION....................................................................40
   7.1 Mode 1 vs. Mode 2 .........................................................................................40
   7.2 Observations ..................................................................................................41
   7.3 Future Enhancements......................................................................................42
   7.4 Conclusion ......................................................................................................43

WORKS CONSULTED .....................................................................................................44
LIST OF TABLES

PAGE

Table 7.1. Corrupted Bits per Stega Bits Sent for Mode 1 vs. Mode 2 .........................41
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1</td>
<td>AWT classes used</td>
<td>9</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Binary symmetric channel</td>
<td>13</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Generator matrix</td>
<td>15</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Flow of steganographic process</td>
<td>18</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Selection of cover media</td>
<td>24</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Cover media as a text file being loaded</td>
<td>25</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Encoding of cover media</td>
<td>26</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Parity check matrix</td>
<td>32</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Insertion of secret message: Mode 1</td>
<td>34</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Decoding of a secret message: Mode 1</td>
<td>35</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Encoding of a secret message: Mode 2</td>
<td>38</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Decoding of a secret message: Mode 2</td>
<td>39</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to thank Professor Carmelo Interlando, Department of Mathematics and Statistics, San Diego State University for guiding me throughout my thesis term. His support and encouragement were instrumental in achieving the optimum results in this thesis. I offer my regards to all of those who supported me in any respect during the completion of the thesis.

Lastly I would like to dedicate this thesis to my father Mr. Ashok A. Dixit and mother Aparna A. Dixit. Without their endless support and faith in me, this work would not have been possible.
CHAPTER 1

INTRODUCTION

Steganography is an art or science of transmitting hidden messages. In modern communication system, this means hiding information in communication media such as audio, text and images. Steganography is supposed to be originated from Greek culture where the Greek word steganos means concealed and graphein means to write.

Techniques for hiding information have existed for centuries. In Ancient Greece, secret messages were written on wooden plates and wax was used to cover them. Methods include writing hidden messages on paper written in invisible ink in the blank spaces of the papers. This technique was adopted quite successfully during World War II by the French. Some other techniques were also implemented in the past. Messages were written on the back of postage stamps. Germans used microdots during World War I and World War II. Microdots are nothing but a text or an image substantially reduced in size onto a disc of 1 mm in diameter. Special cameras were used to generate microdots attached to letters. These microdots usually went unnoticed for any intruders and could easily read by the authorized recipient microscope. Techniques such as spread spectrum are used these days in digital communication. Electromagnetic or acoustic signals generated for a specific bandwidth are spread over a much wider bandwidth to avoid signal interference or signal jamming.

Digital watermarking is also one of the many applications of steganography. Visible watermarks are used for copyright protections and source tracking but in case of invisible watermarking, the information is difficult to perceive. The secret message is hidden in a digital signal. The spread spectrum mentioned earlier is used for audio watermarking. Spread Spectrum is used to embed watermarks which can be implemented easily in any time domain. After spreading the spectrum the information is hidden in the form of a watermark and is added to the sender signal as a watermarked signal.

The core principal for steganography is that apart from the sender and the receiver, no third party or the intruder can suspect the presence of any such hidden or covert message. This phenomenon clearly distinguishes steganography from a very renowned technique of
information hiding which is cryptography. In cryptography, the information is hidden by doing encryption of the original message by various encryption algorithms. This encryption process converts the plain text into a cipher text with the help of the encryption key. If ever a third party intrudes and manages to extract the cipher text, this encrypted message is hard to decode without the key. This clearly states that in Cryptography the third party can detect the presence of secret message easily though it may or may not decrypt the encoded message which is disparate from the principal of steganography which hides the message as well as the presence of the message. Therefore where cryptography protects the content of the message, steganography protects both message and communicating parties. So except the authorized persons, no third party can ever think of any such secret message in the communication. Hence steganographic communications do not attract attention since they are never highlighted or encrypted but always hidden.

In computer systems as well, steganography is extensively used. Pictures are embedded in video material. Secure shell connections, remote desktop software such as telnet, virtual host always include some amount of delay before sending the information packets over the network. These delays can be used to encode data. Texts are hidden in web pages. Information is concealed within computer files which can be audio files, jpeg images or bit mapped images which are larger in size and contain lot of information in it. For example, every $n^{th}$ color bit is replaced with some message bit and sent over the transport network. This change is so minute that it usually goes unnoticed due to highly redundant code stream.

Some tools can be used to transmit valuable data in normal network traffic. Internet Control Message Protocol (ICMP) is an Internet protocol used for networked computers to send error messages for diagnostic or routing purposes in IP datagram. These ICMP messages are part of the IP header and transmitted the resulting datagram. Linux has a ping utility which adds 56 bytes of ICMP message to the existing header. Loki is another such tool that hides data in ICMP traffic. Loki is a client-server program which can be used to transmit data secretly across the network through back door into a Unix system. A directory starting with dot (.) is a hidden directory. A directory starting with three dots (…) can be created to store secret files so they do not come into the file lists. On Windows systems, the C:/winxp/system32 or C:/winnt/system directories are where all the Windows .dll, .dib and
set up files are placed. This directory can be used to securely store all the covert files assuming that no one really dares to tamper or touch the files in those important directories.

As we move forward, Chapter 2 will cover the functional requirements, and Chapter 3 will contain information about the technologies used. Chapter 4 tells more about the error correcting codes, binary symmetric channels, etc. Chapters 5 and 6 explain the encoding and decoding mechanisms in detail, and Chapter 7 concludes with observations and future enhancements.
CHAPTER 2

REQUIREMENTS

The primary aim behind developing this tool is to understand how steganography can be achieved using error correcting codes for very noisy communication channels with cover media being a text file. Upon completion, this tool will help the students from mathematics and statistics as well as computer science in learning more about error correcting codes and their applications and different techniques used in Information security branch for secure and covert data communication. This software can also be integrated into different text editors such as office word, Kwrite etc. for creating documents with secret messages embedded. The requirements gathered have been further classified into platform requirements and functional requirements.

2.1 PLATFORM REQUIREMENTS

- The main objective in choosing the software development kit (software language) was that it should be platform independent so that final product will have the capability to run on any environment irrespective of the operating system.
- The software should run as a stand-alone application rather than a web based software. Hence Java SDK instead of Java Enterprise Edition has been chosen to be the appropriate language for writing the code. The operating system is Ubuntu Linux considering in mind the importance of open source software.
- There should be a facility to store different versions of code and some repository where code can be checked in and checked out. This is applicable whenever any modifications are made or any new feature gets added. Keeping this in mind, SVN (subversion) repository has been used.

2.2 FUNCTIONAL REQUIREMENTS

- The software should be able to read any large text file as a cover media so that secret message can be embedded into it.
- The Error correcting code (ECC) being chosen should not induce additional complexity to the existing bit stream.
- The ECC should not increase the bandwidth of the channel by adding too many redundant bits in such a way that the performance and efficiency gets hampered.
• The core principle of steganography should be achieved. That is, message as well as its existence should be concealed.

• The transportation of the message blocks over the communication channel should not take large amounts of time.

• The decoder version of the software should be capable enough to join all the received and decoded data blocks to get back the original secret message is received.

• Since the Golay code is used as an ECC which can correct up to three errors per 23 bit codeword, there should be a facility for the encoder to select how many artificial errors (stega bits) he wants to send for each codeword.

• In the Golay code mode2, the detection scheme should be intelligent enough to select the erred bits which were not picked by the normal decoding mechanism.

• There should be a facility to introduce the genuine channel errors along with the artificial errors so that the communication channel used will look more original.
CHAPTER 3

TECHNOLOGY

This chapter focuses on the technology used to develop the software and briefly discusses about the supporting software, tools and integrated development environments (IDE).

3.1 JAVA

The entire steganographic software is built using Java Software Development Kit 6. The integrated development environment used for the writing Java classes is Eclipse Galileo. The reasons behind choosing Java over other software languages include the following.

- Java is simple, easy to implement and object oriented.
- Java provides high performance using its very large set of application programming interfaces (APIs).
- Java is robust and secure.
- Java can provide multi threaded programming so that the program execution is faster and it is dynamic.
- Java is platform independent, architecturally neutral and highly interpretable.
- Java has excellent set of Graphical user interface APIs in form of its abstract window toolkit (AWT) class as well Java Swing class.

These points are elaborated below.

An object oriented programming language is one which lets you create objects. An object is an entity which drives the class attributes and functions to which it belongs. An object oriented model is a collection of interacting objects which is different from conventional programming. Java is object oriented because it focuses on creating objects and making them work together. The process of creating an object is known as instantiation. Java features all of the object oriented concepts mentioned below.

- Polymorphism: A single method can generate different set of results when passed with different set of arguments.
- Inheritance: Classes are arranged hierarchically and child class can access methods and attributes of its parent class.
Data encapsulation: The attributes, variables and methods of a particular class are differentiated based on their role in the programming model such as public, private, protected.

Things like these make programming very loosely coupled so that the complexity is reduced and the programming models become highly independent and modular.

3.1.1 Simplicity

Java compiler automatically creates the Java compiled classes into machine readable byte-code. The most important feature of Java which makes it very simple is its ability to handle automatic memory management. Java uses automatic garbage collection when an object is destroyed to release the memory unlike C++ where programmer is responsible for freeing the memory associated with the deleted object.

3.1.2 Robustness

A robust programming language is very stable, secure and does not fall prey to third party trapdoors. Hence it is very reliable. This is because Java is highly supported language, intended for use in networked environment. No programming language can really assure full-proof reliability but there aren't much security holes in Java. An example is a bad Java program will never crash your computer unlike a C program. Java is dynamic in a sense that Java puts lot of emphasis on runtime error checking and eliminating situations which are error prone.

3.1.3 Multithreading

A multithreaded program divides any process into several threads. A thread is a smallest unit of program execution. These individual threads run in parallel to allow faster execution of a program and increase the program execution speed. Java has a separate API dedicated for multithreaded programming which has been smoothly integrated into it unlike C++ where operating system specific procedures have to be called in order to enable multithreading.

3.2 Java AWT Components

AWT is used extensively with Java to provide windowing, graphical support and adding graphical widgets to Java modules. Since Java is platform independent, AWT is also
platform independent intended to provide a common set of tools for graphical user interface design. The elements used by AWT are provided using the native graphical user interface toolkits of a platform on which the AWT is implemented. AWT is now a part of Java Foundation Classes and can be implemented by importing its APIs. The syntax is:

```
import java.awt.*
```

This will import the entire AWT package which has various individual classes for painting graphics and images. Some of the classes used in the thesis are as follows.

- **Frame**: This class provides a frame where all the AWT components are placed. Frame can be resized, hidden or made visible.
- **Button**: This class helps creating labeled buttons which can be associated to various actions using event listener class.
- **Color**: This class allows to paint the user interfaces in default RGB color space or a custom color space.
- **Container**: This class acts like a container which can contain various AWT components.
- **TextArea**: This class creates multiple line objects which permits text typing and display.
- **Dialog**: This class is opens up a new window which can be used to take some form of an input from the user.
- **CheckBoxGroup**: This class is used to group the different check box buttons where only one check box button can be in the active state at a time.

All these AWT classes need some interface for implementation. These interfaces define how to lay out the various components. Some of the interfaces are Paint, LayoutManager in Java. LayoutManager has been used in for creating AWT platform which creates layouts such as grid layout, stream layout or border layout etc. Figure 3.1 shows how the AWT classes have been used in the thesis.

### 3.3 Subversion Repository

Subversion is an open source version control system which has been used as a support tool for this thesis. Subversion provides a repository where one can store the files and it keeps track of all the file modifications, new files addition, files removal based on date, time etc. This feature gives the thesis the flexibility to recover the system into the older versions if something in the current version goes wrong and there is a necessity to get back to the previous version. Subversion uses Apache Software Foundation license, which makes it an
open source system. Subversion once downloaded and installed can be accessed from command line as well as from a subversion client which uses GUI.

Some of the features of subversion include:

- The transactions are atomic, i.e. if anything goes wrong while code check-in or checkout to the repository, the transaction is rolled back to its original state.
- Subversion repository can be loaded onto the storage server which makes it more flexible as various people across the network can use it simultaneously.
- Binary differencing algorithm is used which works equally well for normal text files, Java classes as well as executable binaries.
- Branching or tagging a file is not expensive operations as they consume much less time and are totally independent of the file size.
- Subversion can be used as a very effective tool in Software Configuration Management.

To add a new stream or file, one uses the command `svn add -filename/directory` to add the entire directory or a file to the repository. This is done for the very first time when a new file needs to be added. The next step is to commit the file to the repository with the
command `svn commit - filename/directory`. This action is performed for all the subsequent times to make revisions to the file/directory. To delete a file or a directory, the following command is used: `svn remove - file/directory --force`. 
CHAPTER 4

NOISY CHANNELS AND ERROR CORRECTING CODES

Any message to be transmitted over the communication channel needs some level of protection. This is because of many things such as noise, channel error in the communication. These hindrances not only change the message content but also the meaning of the message very commonly known as noise.

4.1 NOISE

Noise is an unwanted part of any digital or analog signal. It is the factor which is responsible for degrading the quality of the signal by acting as an interference or blockage in the communication channel. This entity is naturally present in most of the communication channels corrupting the signals passed over. Hence signal to noise ratio should be as high as possible to ensure the error free communication. However the occurrence of noise is totally random and if proper filter is not used to detect its presence, it usually goes unnoticed creating disturbances at the receiver's end. To combat the noise, various techniques are used such as increasing the power of the signal, implementing some sort of a modulation such as frequency modulation (FM), amplitude modulation (AM) etc. or adding a lot of redundant bits to the original signal. Some of these operations are expensive such as increasing the power of a signal or amplitude modulation. Adding a lot of redundant bits can also be an inefficient method since it increases the channel bandwidth beyond capacity. But if handled in a proper manner, through error correcting codes this method can be used very effectively for to detect the noisy bits or errors at the receiver's end.

A very noisy channel is an attractive medium for steganographic communications. This is achieved by having a low transmission rate of concealed messages sent block by block. This type of communication produces an almost untraceable secret message transfer. Since these noisy channels require error correcting codes to come over the noise, the code redundancy is utilized very ingeniously to insert the secret message bits to be passed in form
of artificial channel errors. The insertion of steganographic bits over a honest communication channel is possible only because of the bit redundancy created by the error correcting codes.

Error correcting codes (ECC) provide a technique for data transmission in which a few extra bits are added in each block of data in order to detect the errors and then correct those errors that may occur in the communication. These redundant bits also known as parity bits make sure that the message is received error free at the receiver's end and once the errors are corrected, these parity bits are easy to remove from the original content. Parity Bits take care of the number of checked bits in the code i.e. they are implemented as odd parity when number of 1's in a given set of bits are even and similarly for even parity.

ECC have been successfully used in physical and data link layer of the OSI model and also implemented in data disks and computer physical memories in the form of checksum. Checksum is an addition of all the codewords in a given set of bits. ECC are mainly divided into two classes viz. convolution and block codes. Convolution codes operate on bit by bit basis and block codes are processed per block basis. This thesis is focused on block codes. ECC are also known as forward error correction (FEC) since no re-transmission of the message is performed. There are many types of block codes such as repetition codes, BCH codes, Golay codes, Hamming codes. Some of these are very efficient codes such as Golay, BCH etc. Golay Codes are encoding and decoding techniques are implemented in this thesis work. The channel used for steganographic communication is a binary symmetric channel explained below.

4.2 BINARY SYMMETRIC CHANNEL

The communication channel used in most of the steganographic communications and encoding-decoding mechanisms of Golay codes in this thesis is a binary symmetric channel (BSC). Let us assume that the communication messages are sent over a very noisy channel and that we can send only two symbols viz. 0 and 1. Also assume that when the sender sends the symbol 0, the receiver receives the same symbol 0 with probability p and receives 1 with probability q. Similarly when the sender sends symbol 1, the probability that 1 is received is p and 0 is received is q. Then for a binary symmetric channel, \( p + q = 1 \) also meaning \( q = p - 1 \).
This type of communication is possible only in binary symmetric channel (BSC). BSC is very frequently used in information and coding theory. It is assumed that in BSC, the symbol is received with a high probability, but if the symbol or the bit gets flipped, then that probability is very small. Figure 4.1 illustrates this type of communication channel very clearly. In this diagram, if X is a variable that is randomly transmitted and Y is the variable randomly received, then the channel is characterized by the conditional probabilities shown below (Equations 4.1 – 4.4):

\[
\begin{align*}
\Pr(Y = 0 / X = 0) &= p \\
\Pr(Y = 1 / X = 0) &= 1 - p \\
\Pr(Y = 1 / X = 1) &= p \\
\Pr(Y = 0 / X = 1) &= 1 - p
\end{align*}
\]

![Figure 4.1. Binary symmetric channel.](image)

### 4.3 Linear Codes

A linear code of length n and dimension k is a subspace C of the vector space \((F_2)^n\) (all n tuples with entries in \(F_2\) where \(F_2 = \{0, 1\}\) the binary field). Such a code is referred to an \((n, k)\) code. Elements of the code are called codewords. A generator matrix for an \((n, k)\) code is a \(k \times n\) matrix whose rows form a basis for the vector space C. This generator matrix often denoted as \(G\), is of the form \((I_k \mid A)\) where \(I_k\) is an identity matrix \((k \times k)\) and \(A\) is the standard matrix with dimensions \(((n-k) \times n)\).

The Hamming distance of a linear code \(C\) is equal to the minimum distance between any two codewords in \(C\). This is also equal to the minimum weight of the nonzero codewords in \(C\). These linear codes bear a special property that any two non zero codewords \(c \in C\) differ
in at least $d$ positions where $d$ is the Hamming distance between the two codewords and addition of any these codewords, e.g. $c_0$ and $c_1$, fetch a result say $c_2$ which also is a codeword belonging to the same set $C$. In mathematical terms, Hamming distance is the number of coordinates $I$ which $c_0$ and $c_1$ disagree.

### 4.4 Golay Codes and Applications

Golay codes belong to the class of ECC which are used in mathematics and computer science fields extensively. Golay codes were invented by Marcel J. E. Golay. Golay was a Swiss mathematician. Golay codes over the years have played an important role in both the theory and practice of ECC. ECC are generally defined over the finite field namely galois field which contains limited number of elements used in digital communications. It is denoted by $GF(n)$ or $F_n$ where $n$ is the maximum number of elements that can belong to the respective finite field. For every prime integer $p$ and positive integer $n$, there exists a finite field with $p^n$ elements. With $n = 2$, the galois field becomes binary and only the two specified elements can be present in each of codeword belonging to $C$. In most of the cases, $F_2 = \{0, 1\}$ were ‘+’ and ‘.’ are addition and multiplication modulo 2.

Golay codes are divided into two branches, namely binary and ternary. Binary Golay codes use 0 and 1 in $GF(2)$ and they are further classified into extended binary Golay codes (EBGC) or perfect binary Golay codes (PBGC). This thesis is designed for PBGC which is most commonly used in practice and often denoted as just Golay codes.

In case of PBGC, the codeword has a length 23. They are denoted as $[23, 12, 7]$. The numbers in the brackets are explained below. A block of twelve binary bits is converted into a block of 23 bits after passing through a Golay code encoder. Due to the binary nature of a 12 bit message block, the total number of codewords this vector space contains is $2^{12} = 4096$. Any two codewords belonging to this 12-dimensional subspace $W$ of the vector space $V$ differ in at least 7 positions or equivalently, any non zero codeword has at least seven 1’s in it or the Hamming distance between any two codewords is 7. By adding just one parity bit to every encoded 23 bit codeword, the codeword becomes 24 bits and the perfect Golay codes are transformed into extended Golay codes. Extended binary Golay codes denoted as $[24, 12, 8]$ are very similar to perfect Golay codes except the fact that any Hamming distance between any two codewords is 8.
Implementing the encoding and decoding mechanisms of Golay codes is simple but very efficient. The message bits are divided into blocks of 12 bits each. Every such block \( M \) of 12 bits is converted into a block of 23 bits by multiplying it with a generator matrix of \((12 \times 23)\). This generator matrix has its rows 12 different and non-zero Golay codewords which differ from each other in at least seven positions. An example of one such generator matrix is shown in Figure 4.2.

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1
\end{pmatrix}
\]

**Figure 4.2. Generator matrix.**

The subsequent multiplications and additions which occur in \( M \times G \) are modulo 2. Hence the resulting row matrix of 23 bits contains only binary numbers 1 and 0. In the decoding process, the received vector \( r \) of length 23 is possibly corrupted in up to three coordinates. Matrix \( r \) is multiplied with the transpose of a matrix \( A \) to get the vector \( s \). Matrix \( A \) has 506 rows and 23 columns whose rows are dual of Golay codewords. The dual of a linear code \( C \in (F_2)^n \) denoted as \( C^\perp \) is defined as follows: \( C^\perp = \{ u \in (F_2)^n \mid u \cdot v = 0 \text{ for all } v \in C \} \) where \( u \cdot v \) is the usual scalar product between \( u \) and \( v \) and the dimension of \( C^\perp \) is equal to \( n-k \). All the additions and multiplications are modulo 2 again.

The resulting vector \( s \) (1 x 506) is multiplied by matrix \( A \) (506 x 23) to get the vector \( v \) (1 x 23). In this case the additions and multiplications are the usual integer additions and multiplications. Hence 1 + 1 + 1 becomes 3 and not 1 as in the case of modulo 2 operations. The error pattern is determined for each of the 23 bits of \( v \) and stored in \( e \) where \( e_i \) is the \( i^{th} \) position of \( e \) for \( i = 1, \ldots, 23 \). For each position \( v_i \), for \( i = 1, 2, \ldots, 23 \) in \( v \), if \( v_i = 176, 120, \) or 96, then the corresponding bit is in error. This makes \( e_i = 1 \). In all other cases \( e_i \) is 0.
After determining the error vector e, it is added to the received vector r in modulo 2 so that the erred bits are flipped to get back the correct code that was originally sent. Let us consider an example where the 12 bit binary stream 111000111000 was converted to 23 bit Golay codeword 1110001110001111011111 after multiplying with the generator matrix G. Assume 2 bits were corrupted at 11\textsuperscript{th} and 18\textsuperscript{th} position respectively and hence the stream which the receiver received was 11100011101011111111111. When the decoding techniques are applied, the error matrix which is generated is nothing but 00000000001000001000000 contain 11\textsuperscript{th} and 18\textsuperscript{th} bits as 1 and rest others as 0. This error matrix e when gets added to the received vector r yields the original codeword which is 11100011100011111011111. These encoding and decoding procedures are explained in much detail with programming implementation examples in Chapter 5 and Chapter 6 respectively.

Extended binary Golay codes (EBGC) can correct up to four errors and detect up to seven errors whereas PBGC can correct up to three errors and detect up to six errors in a 23 bit codeword. This rate of error correction is very high compared to other ECC hence they were used extensively in communication channels and spacecraft programs in the late 1980s. Golay Codes with proper combination of codewords reduce the amount of noise in the channel which magnifies the signal to noise ratio making the code useful for biomedical Doppler applications. Binary Golay Codes have also been used in NASA spacecraft mission. In the 1980s when the channel bandwidth was very limited, hundreds of thousands of high resolution colorful images of planets such as Jupiter and Saturn were sent using Golay Codes Encoding due to the high probability of error free receipt of these images. EBGC [24, 12, 8] were used in this case since color images required to send 3 times the amount of data and these codes can correct up to three errors. In high frequency radio systems communications, EBGC have been used for forward error correction according to American government standards.
CHAPTER 5

ENCODING OF STEGANOGRAPHIC MESSAGES

As explained in the previous chapters, error correcting codes add bit redundancy to the original message block. This redundancy is added to make sure that bits that might get corrupted during the communication over the noisy channel are received without any error. This bit redundancy is utilized for steganographic purposes where the secret message bits can be passed. These secret message bits are known stega bits which are replaced with redundant channel bits. Also the channel used is binary symmetric channel (BSC). A very noisy channel will have lot of genuine errors, i.e. errors that have caused just because poor channel reception without any steganographic intervention. To differentiate these errors from the artificially inserted stega bits, the stega bits are known as artificial channel errors. The two basic modes that have been implemented are discussed be.

5.1 FLOW OF ENCODING MECHANISM

In MODE 1, the stega bit is inserted in a fixed position within a codeword \( c \in \mathbb{C} \) as follows:

- If the selected bit in the codeword is 0 and the stega bit is also 0, then the bit is not inserted into the codeword. Same would be the case if both the stega and codeword bits are 1.
- If the selected bit in the codeword is 1 and the stega bit is 0, then 1 in the codeword is replaced by 0 stega bit.
- If the selected bit in the codeword is 0 and the stega bit is 1, then 0 in the codeword is replaced by 1 stega bit.

In MODE 2, the stega bit is inserted in a random position within a codeword \( c \in \mathbb{C} \) as follows:

- 0 stega bit is inserted in a codeword as an artificial error in such a way that it replaces any randomly selected 1 in a codeword \( c \in \mathbb{C} \).
- 1 stega bit is inserted in a codeword as an artificial error in such a way that it replaces any randomly selected 0 in a codeword \( c \in \mathbb{C} \).
- If the codeword does not contain any position occupied by 1 and 0 is to be inserted, then no stega bit is inserted.
• If the codeword does not contain any position occupied by 0 and 1 is to be inserted, then no stega bit is inserted.

• In both modes, code C is used to recognize both the error bit position as well the erred status.

Thus the decoding mechanism should be intelligent enough to find out the stega bit positions. Golay Codes, Hamming Codes or Repetition Codes have excellent decoding mechanisms to find out the bits in errors and correct them. However the separation of genuine errors from the artificial errors is a difficult task which is handled by separate techniques in both the modes.

In case of Mode 1, the stega information is encoded in a known position hence at the decoding end, it is quite easy to determine which bit is in the error and then to check the error status. In case of Mode 2, the stega information is carried by a bit which is randomly encoded in a codeword unknown to the receiver, hence it becomes very difficult to identify the erred bit at the receiver end just with the help of normal decoding mechanism. Therefore along with this decoding mechanism, special detection criteria must be defined.

The entire process of encoding and decoding the stega message to pass over the steganographic channel (which is BSC) is shown in Figure 5.1. S is source and U is a user. E is a binary encoder which uses a liner code C, a BSC with error probability of p and Decoder D. A decoding rule is for the corrupted codewords is described by the full set of coset leaders T and encoding mechanism converts the Codeword into a encoded bit stream K passed over stega-channel.

![Flow of steganographic process.](image)

**5.2 Repetition Codes**

Repetition codes are a type of error correcting codes. They are denoted as (r, 1) where r is the repetition index, which makes every bit of the secret message repeat by the repetition
index r. Hence they are called repetition codes. For repetition codes, each codeword is of length r. For example if the signal \( s = 10100 \) and the scheme used is \((3, 1)\) then every bit of s is repeated thrice and sent over the communication channel. Hence the extended codeword becomes \( c = 11100111000000 \).

An encoder for a repetition code is a simple device which repeats every bit of the information sequence by the repetition index. This method of encoding is very simple but quite unsophisticated. As a result, it is hardly used in practical applications. But it can give a nice insight about the steganographic process for ECC and also which method of ECC is to be used based on efficiencies. At the receiver end for every bit stream of length r, the codeword is compressed into a single bit using the majority element rule. For example, consider the codeword received is 11000111000001. In this case since r is 3, every set of three bits is compressed into a single bit using this majority rule which simply counts the number of occurrences of 1 and 0 and sets the output bit according to the bit in majority. Here:

- 1st 3-bit block is 110, hence the majority element is 1.
- 2nd 3-bit block is 000, hence the majority element is 0.
- 3rd 3-bit block is 111, hence the majority element is 1.
- 4th 3-bit block is 000, hence the majority element is 0.
- 5th 3-bit block is 001, hence the majority element is 0.

All five bits are concatenated to get the decoded message which is 10100. Hence at the receiving end, the user receives original signal which is 10100.

Repetition codes offer a poor solution for the large files datasets since they repeat every bit by r times and increase the channel bandwidth beyond its capacity. The only advantage of repetition codes is that their execution is fairly simple. Use of repetition codes for steganographic communication is explained in the following example.

Consider a secret message as 10100010 and cover media which can be a text file or an image be 11010000. We are using repetition codes with scheme \((3, 1)\), i.e. \( r = 3 \) which makes the encoder repeat every bit of cover media by 3. This operation generates the elongated version of cover media X as 111111000111000000000. Every first bit of the 3 bit block is ex-or with one bit each of secret message as follows:
\[
X = 111\ 111\ 000\ 111\ 000\ 000\ 000\ 000
\]

\[
X\text{-OR}
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
011 & 111 & 100 & 111 & 000 & 000 & 100 & 000 \\
\end{array}
\]

This new \( X' \) which is 011111100001000000100000 is transmitted over a binary symmetric channel. Lets suppose that this noisy channel introduces few errors in \( X \) and the message received at the User end is \( Y = 01111100011000100000 \) with the marked bit as a corrupted bit due to genuine channel error. A majority element array is calculated for this received codeword considering majority rule for every three bits. The decoding operation is described as follows:

\[
Y = 011\ 111\ 100\ 111\ 001\ 000\ 100\ 000
\]

\[
R = 111\ 111\ 000\ 111\ 000\ 000\ 000\ 000
\]

\[
X\text{-OR}
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]

\[
Y' = 100\ 000\ 100\ 000\ 001\ 000\ 100\ 000
\]

For \( Y' \), every 1\textsuperscript{st} bit of the three bit block is fetched to get the secret message. The secret message received is 10100010. The secret message that was sent was 10100010. Hence we have got the exact same secret message despite the fact that there were some artificial errors i.e. stega bits and genuine channel errors. This technique worked for the small codeword but looking at the bigger picture, a color image with RGB as (8 bit *3) 24 bit image would be converted to a 72 bit stega channel which would be very inefficient. Hence from now on, the steganographic communication is achieved using Golay codes which are more powerful than repetition codes. Due to their simplicity, the repetition codes are currently used in following applications:

- Some Universal Asynchronous Receivers and Transmitters use majority filters to ignore modulations in the noise known as noise spikes. This spike injection filter is a repetition decoder.
- Many frequency modulation techniques in the current world transmit a single bit or a block of few bits over many sinusoidal signal cycles. The low pass filter used at the receiver's end for the entire bit stream is assumed to be a repetition decoder.

### 5.3 Golay Codes Encoding

Encoding of Golay codes has been explained in brief in the previous chapter. The programming details and implementation techniques that were used in this thesis are
described below. Golay codes can detect up to six errors and correct up to three errors in a codeword of 23 bits. The example considered here has a cover media as a text file and secret message as a plain text. Both the cover media and the secret message are converted into binary streams of 1 and 0. This conversion takes place using a function named readFile(). This function has been written in Java and explained below.

The readFile() command takes its argument as the filename whose contents are supposed to be converted and stored into a binary stream. This stream is created by a Java class which opens the file, reads it and makes use of Java input-output functions by importing Java.io api. The large stream is first read into a variable which contains non-binary characters. This string is separated into individual characters and each character is stored into a character array as an array variable using toCharArray() function. Hence if the string is Desktop, the character array, e.g. arr[], is created and Desktop is stored into arr[] as arr[0] = 'D', arr[1] = 'e', arr[2] = 's' etc using the toCharArray() function.

Each character has an associated integer value with it. This integer is also known as an ASCII value. For example, 'D' has an ASCII value 68, 'A' has ASCII value 65, 'B' is 66, 'e' is 101, number '0' is 48 and Space is 32. The character array elements are converted into the respective integer values using typecasting technique of Java. Typecasting changes the data type of a variable to another data type. Here a character is converted to integer by simply specifying the character as (int) character, which converts it into an integer value. This integer value is further converted into its respective binary value which ultimately we want using a direct toBinaryString() function which Java provides.

Now 'A' <=> 65 which when converted to binary fetches 1000001 is a 7 bit binary stream. Another value, e.g. number '6', <=> 54 when converted to binary stream fetches 110110 which is a 6 bit binary stream. When entire message stream needs to be converted into a binary stream, all these small binary streams are concatenated. At the decoder end, it would be fairly difficult for the decoder to divide the received bit stream back into ASCII values since the length of individual characters in binary are different as we just saw. To overcome this problem, all the characters are converted to 8 bit binary streams by appending zeros at the start. Hence 'A' becomes 01000001 by appending one zero and '6' is converted to 00110110 by appending two zeros at the start. The entire process of starting with a string and fetching an 8 bit binary stream is explained in the following function:
public int[] readFile(String passedString)
{
    tempArray = new char[passedString.length()],
    tempArray2 = new char[tempArray.length * 8],
    intempArray = new int[tempArray.length],
    longInt = new int[tempArray2.length],
    String finalString = "",
    for(int i=0;i<longInt.length;i++)
    { longInt[i] =0, }
    String str7 = "0",
    String str6 = "00",
    tempArray = passedString.toCharArray(),
    for(int i=0;i<tempArray.length;i++) {
        intempArray[i] = (int)tempArray[i],
        strBinary = Integer.toBinaryString(intempArray[i]),
        if(strBinary.length()==7){
            strBinary = str7.concat(strBinary),
        }
        else if(strBinary.length()==6){
            strBinary = str6.concat(strBinary),
        }
        finalString = finalString.concat(strBinary),
    }
}

All the small binary sub-streams are concatenated and stored into a final large stream declared as finalString shown above. For storage purposes, the finalString is stored into a character array where each character is either 1 or 0. This entire character array is converted into an integer array (not into the ASCII integer values but just the regular integer values) where character '1' gets replaced with integer 1 and character '0' is replaced with integer 0.

Thus a large binary string of cover media is generated at the sender using the above technique. This binary string is now divided into blocks of 12 bits each. Remember that Golay codes encode 12 bit block into a 23 bit codeword using encoding mechanism. This 12 bit block is first transformed into a row matrix with twelve elements, that is, if X = 101000101110, then X is converted into a row matrix as shown in Equation 5.1:

\[
M = \begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0
\end{bmatrix}
\] (5.1)

M is multiplied to the generator matrix G of Golay codes which have 12 rows and 23 columns. The matrix multiplication (M x G) produces a matrix A, a row matrix with 23 bits. This row matrix is one of the 4096 codewords of the Golay codes. As explained
before, all these subsequent multiplications and additions are modulo 2. Hence matrix A will contain only binary numbers 1 and 0. The conversion from 12 bit block to 23 bit codeword is processed for all the subsequent binary blocks of stream created of cover media text file and all these 23 bit codewords are appended to generate a large stream of bits. This conversion and matrix multiplication is explained below in function matMultiply():

```java
public int[][] matMultiply(int[][] A, int[][] B) {
    int C[][] = new int[A.length][B[0].length],
    for(int i=0,i<A.length,i++) {
        for(int j=0,j<B[i].length,j++) {
            C[i][j] = 0,
        }
    }
    for (int i = 0, i < A.length, i++) {
        for (int j = 0, j < B[i].length, j++) {
            for (int k = 0, k < A[i].length, k++) {
                C[i][j] = ((C[i][j]+(A[i][k] * B[k][j]))%2),
            }
        }
    }
    return(C),
}
```

This function takes its arguments as two matrices supposed to get multiplied viz. matrix A and matrix B. The result is stored into matrix C. For matrix multiplication, the number of rows of (B) must equal the number of columns of (A). Also the resulting matrix C should have dimensions such that there are the same number of rows of (C) as (A), and the same number of columns of (C) as (B).

Hence the matrix C is declared as “int C [][] = new int[A.length][B[0].length],” where A.length specifies number of rows of A and B[0].length specifies number of columns of B. The matrix C is initialized to zero. The matrix multiplication follows the following rules:

- Every element from the row of matrix A is multiplied to its corresponding column element of the matrix B. For example, the 1st row, 3rd column element of matrix A is multiplied by the 3rd row, 1st column of matrix B and so on.

- All the intermediate multiplications are added for each row of A and each column of B and the resulting element is stored in the respective matrix dimensions. In short, every row of matrix A is multiplied to every column of matrix B and resulting element is stored at row number of A and column number of B in the resulting matrix.
• All these multiplications & additions are modulo 2. This is emulated in the following equation 
\[ C[i][j] = ((C[i][j]+(A[i][k] * B[k][j]))\%2) \]
where k denotes columns of A which is equal to the rows of B and i and j denote rows of A and columns of B respectively.

• After the matrix multiplications are done, all the resulting 23 bit codeword streams are combined and stored a stream into an array. This array is converted back to ASCII characters in order to print the encoded string of bits on the user interface.

The operation of opening up the text file using file reader dialog box, loading the file, and applying the Golay codes encoding process is explained below.

As shown in Figure 5.2, the file selection dialog box is opened after clicking “Select a text file” button. Here in this case the selected file is sendfile.txt which is located in the directory /home/aniket/Desktop/Thesis. This sendfile.txt is loaded into a text area as shown below. The text in this file is now open for conversion into a binary stream and from there to encoded binary stream of Golay codewords. This operation is initiated when “Encode File” button gets pressed which is placed just below the text area.

![Golay Codes Steganographic Tool](image)

Figure 5.2. Selection of cover media.
The contents shown in Figure 5.3 are transformed into a binary stream as explained before. This binary stream is converted into an elongated binary string when it is divided into blocks of 12 bits each and passed through the encoder to receive 23 bit stream per block. To get a better idea of how this elongated binary stream with all the redundant bits looks like, it is converted back to the actual text with the help of reverse ASCII operation which is shown below. The cover media now looks a series of garbled characters which can act as a perfect steganographic cover for the secret message. A naked eye can barely observe the difference between such a noisy looking channel with steganographic message embedded into it and the noisy channel without the steganographic message embedded into it. Figure 5.4 clearly demonstrates this point.

Once the cover media is encoded, the sender has the privilege of using one of the two modes discussed above viz. Mode 1 and Mode 2. The sender can also decide upon how many number of secret message bits he wants to send per codeword. Based on the selection criteria, the fixed number of bits at certain or random positions is replaced; this is discussed in the next section.
5.4 INSERTION OF STEGA BITS INTO THE COVER MEDIA

The process for Mode 1 and Mode 2 are discussed separately below.

5.4.1 Mode 1

Let us assume that the user selected Mode 1 encoding option. The basic rule of Mode 1 encoding is that the bits at fixed positions are replaced with stega bits. The number of stega bits user wants to insert per codeword depends upon the option which he has selected from the radio buttons in the picture above. Hence if a user selects Mode 1 with 2 bits to encode
per codeword, then 2 bits in each codeword at some fixed position lets say 1 and 11 are replaced with secret message bits.

To perform this operation based on the number of stega bits to be replaced, the secret message bit array is further subdivided into small sub-arrays. That is if 1 stega bit per codeword is to be inserted, then small sub-arrays of length 1 are created from secret message stream. If a user wants to replace 2 bits per codeword then the large secret message bit stream is divided into pieces of length 2. This is done with the help of following Java statement 'int [][] split_secret = new int[1][stega]' where stega is the number of stega bits passed dynamically. Bits at the following positions are replaced for various options.

- For 1 stega bit encoding, 2\(^{nd}\) bit of every codeword is replaced with 1 bit sub-array.
- For 2 stega bit encoding, 2\(^{nd}\) and 11\(^{th}\) bits of every codeword is replaced with 1\(^{st}\) and 2\(^{nd}\) bits of 2 bit sub-array.
- For 3 stega bit encoding, 2\(^{nd}\), 11\(^{th}\) and 19\(^{th}\) bits of every codeword is replaced with 1\(^{st}\) 2\(^{nd}\) and 3\(^{rd}\) bits of 3 bit sub-array.

This is explained in the following code:

```java
if(stega == 1){
    matrixA[0][1] = matrixB[0][0],
}
if(stega == 2){
    matrixA[0][1] = matrixB[0][0],
    matrixA[0][11] = matrixB[0][1],
}
if(stega==3){
    matrixA[0][1] = matrixB[0][0],
    matrixA[0][11] = matrixB[0][1],
    matrixA[0][19] = matrixB[0][2],
}
```

This snippet is a part of the function insertStegaMode1 which takes its parameters as the 2 arrays, 1\(^{st}\) the 23 bit codeword and 2\(^{nd}\) the dynamically created matrixB which is nothing but the split_secret array declared above. This function also takes its arguments the number of stega bits to encode and it is stored in a variable stega which simply replaces the bits based on the option set.

### 5.4.2 Mode 2

Mode 2 Encoding is bit complex. Here the bits at random positions of a codeword are replaced with that of the secret message. For 1 stega bit per codeword encoding, a random
number between 0 to 22 (23 bit codeword) is chosen using function nextInt(n) of java.util.Random class which generates an integer between 0 and n-1, boundary values included. Since it is a 1 stega bit encoding operation, 1 bit sub-arrays of secret message array are created as explained above. This 1 bit array element replaces the codeword bit at the randomly selected position.

This operation works very well when only one bit is to be replaced, but when more than one element is to be chosen randomly, the algorithm becomes tricky. Here the nextInt(n) function is used more than once. Lets say while generating two random numbers between 0 and 22, nextInt(22) fetched 15 first and then 11. In this case, the 16th and 12th (0 is the first bit) bits are replaced with bits of dynamically created 2 bit array. The 23 bit encoded array is now sent over the channel very well. But at the receiver end, the decoding mechanism would fetch 12th bit first and then the 16th bit since the decoder is totally unknown of the sequence in which the random numbers are generated. Since this sequence of bits replacement gets lost at the receiver end, the decoding generates a wrong stream even though it finds the random positions. Hence following mechanism is adopted for multi bit Mode 2 encoding.

For N stega bits encoding, the 23 bit stream is divided into N sub streams of equal length each 23/N. A random number is calculated for every sub stream using nextInt(N). In every preceding sub stream except the 1st sub stream, the bit at (nextInt(N) + (23/ N)) position is replaced with the Nth bit of dynamically created N bit message array.

For example, if N is 2, the 23 bit stream of cover media is divided into sub streams of 11 and 12 bits respectively. The secret message array is further divided into small sub arrays of length 2. For 1st sub stream of 11 bits, a random position is calculated viz. nextInt(11). Let us say this operation created an index 6. Then the 7th bit of 11 bit sub stream replaced with the 1st bit of the 2 bit secret message array. For the 2nd sub stream of 12 bits, a random position is calculated again using nextInt(12). Suppose this gives an integer 10. Then according to the formula stated above, the expression given in Equation 5.2 is replaced by the 2nd bit of the 2 bit secret message array:

\[(10 + (23/2)) = (10 + (11)) = \text{i.e. 21st bit}\]  \hspace{1cm} (5.2)

The same pattern is followed for 3 stega bit operation as well. The Java code is shown below for this operation:

```java
int min1 = 11,
```
int min2 = 22,
int y1 = r.nextInt(min1),
int y2 = r.nextInt(min2 - min1 + 1) + min1,
if(matrixA[0][y1] != matrixB[0][0]){
    matrixA[0][y1] = matrixB[0][0],
}
if(matrixA[0][y2] != matrixB[0][1]){
    matrixA[0][y2] = matrixB[0][1],
}

These small sub arrays are joined to get back the 23 bit codeword with message bits encoded. The 1 stega bit or 2 stega bits encoding operation is repeated for the entire length of the secret message and the final bit stream of secret message encoded stream is passed over the communication channel with few of the genuine errors introduced to the receiver.
CHAPTER 6

DECODING OF STEGANOGRAPHIC MESSAGES

The encoding mechanism explained in the previous chapter showed how the steganographic message was secretly embedded into a cover media, a text file in our case. While encoding the secret message, all the binary bit blocks were either encoded using either Mode 1 or Mode 2 based on the option selected by the user. The decoding process involves applying the decoding mechanism of Golay codes for each of the 23 bit block, then tracking the bits in error and retrieving the original message back which was secretly encoded.

In case of Mode 1 encoder, bits at known position in a codeword are replaced whereas in Mode 2 encoder, bits at random position are replaced with secret message bits. Mode 1 decoding is very straightforward as both the parties are in accord for the encoded bits to look out for. The number of bits corrupted is also communicated to the receiver. Mode 2 decoding can be a difficult task as sender and receiver are completely unaware of the position of the erred bits. The only thing the Mode 2 receiver knows is the number of expected errors in each block. These numbers also go wrong a number of times as the genuine channel errors can corrupt the cover media very badly. Hence for these very noisy communication channels, whenever Mode 2 is used one needs some additional mechanism to detect the errors.

6.1 PARITY CHECK MATRIX

Whenever the encoded bit stream is passed, an additional metadata such as the mode used as well as the number of stega bits used per block are also passed along with the stream. This data can be a part of the stream which gets encoded as well and is appended at the end of the string. In Java program implementation however, this information is sent as a part of parameterized constructor of DecodeGolayCodes() class. An example of such execution is:

```
DecodeGolayCodes decode = new DecodeGolayCodes (encodedString, mode, stega_bits).
```

Once the decoder receives the stream of bits, it is broken into pieces of 23 bits each. Each block of 23 bits is called a word. This word is multiplied with the transpose of a special matrix having 506 rows and 23 columns. A transpose of any matrix A is denoted as $A^T$. 
where $A^T$ is created in such a way that rows of $A$ become columns of $A^T$ and columns of $A$ become rows of $A^T$. In simple terms, for any element of $A$ at $(i, j)$ position it will be at $(j, i)$ position in $A^T$. From a programming perspective, this operation is performed as explained below.

Consider a matrix mat2[][] whose transpose is to be calculated and stored into another matrix transpose_mat2[][].

```java
int[][] transpose_mat2 = new int[mat2[0].length][mat2.length],
for (int i=0,i<mat2.length,i++) {
    for (int j=0,j<mat2[0].length,j++){
        transpose_mat2 [j][i] = mat2[i][j],
    }
}
```

Here mat2.length = rows of matrix mat2 and mat2[0].length are the columns of mat2 which are set for columns and rows of transpose_mat2 respectively. After that every mat2[i][j] element is set to transpose_mat2[i][j]. The parity check matrix shown is nothing but a dual of Golay codewords. A dual of a code has been defined in Chapter 2 which states that in a coding theory, the dual of a linear code $C \in (F_2)^n$ is defined in Equation 6.1 as,

$$C^\perp = \{ u \in (F_2)^n \mid u \cdot v = 0 \text{ for all } v \text{ in } C \}$$ (6.1)

where $u \cdot v$ is the usual scalar product between $u$ and $v$ and The dimension of $C^\perp$ is equal to $n - k$.

This special matrix is nothing but a parity check matrix of a generator matrix $G$ which is defined at the time of encoding algorithm. Parity check matrix is defined as a matrix for a linear code $C$ is an $(n-k \times n)$ matrix whose rows form a basis for $CT$. Property. Given $c \in (F_2)^n$, $c \in C$ if and only if $cH^T = 0$. The parity check matrix is shown in Figure 6.1.

Any two elements of this matrix differ in at least eight positions, a property suited for parity check matrix. A secret message embedded cover media that is broken into blocks of 23 bits each. Every such block is stored into a row matrix $R$ multiplied with this huge parity check matrix denoted by $A$. According to the matrix multiplication rules explained in the previous chapter, this operation yields a matrix $S$ which has number of rows that of cover media matrix and number of columns that of parity check matrix. Hence codeword matrix $R$ with dimensions $(1 \times 23)$ when multiplied to a transpose of parity check matrix $A^T (23 \times 506)$ yields a row matrix $S$ with 506 entries, a very large row matrix. All additions and multiplications in this process are modulo 2 so the output matrix contains either 0 or 1.
Matrix multiplication has been explained in the previous chapter; from a programming point of view, this matrix gets stored into a 2-dimensional array as follows in Equations 6.2 through 6.4:

\[
\begin{align*}
\text{int } S[\cdot]\cdot[\cdot] &= \text{new int}[1][506], \\
\text{int } \text{mat1}[\cdot]\cdot[\cdot] &= \text{new int}[1][23], \\
S &= \text{matMultiply(mat1,transpose_mat2)},
\end{align*}
\]

where matMultiply is the function used for matrix multiplication having all calculations as Modulo 2. Here mat1 is the received vector R, broken into 23 bit blocks and transpose_mat2 is the transposed version of parity check matrix A, denoted by \(A^\top\).

### 6.2 Mode 1 Decoding

After storing the result in S, this large row matrix S (1 * 506) is multiplied with the parity check matrix A (506 * 23) to yield a resultant matrix (1 * 23) stored in V. Note that V has the same length as any normal codeword would have. This particular multiplication however is not modulo 2 multiplications. Hence S would contain all the possible integers except 0 and 1. Also the subsequent additions and multiplications not being modulo 2, operations such as \(1 + 1 + 1\) would yield 3 and not 1. If these two multiplications are combined, then the entire operation can be pictured as follows.

- \(S = R \ast A^\top;\)
- \(V = S \ast A;\)
• Hence \( V = R \times A^T \times A; \)
• \( V = R \times I \) (since \( A^T \times A \));
• Hence \( V = R \) with error bits marked.

Since one multiplication is modulo 2 and other is not, the resultant vector \( V \) would not be equal to received vector but the errors will be clearly highlighted in \( V \). An error matrix \( E[i] \) is generated for \( V \) having equal length as \( V \) in such a way that if \( V[i] = 96, 120, 176 \) then \( i^{th} \) bit is marked as an error. These numbers if occur in \( V \), the respective bit in \( E \) is marked as 1 otherwise it is 0. If \( V[i] = 176 \), then only single bit in the entire codeword is erred. If \( V[i] = V[j] = 120 \), then 2 bits in \( V \) are corrupted. Otherwise, if \( V[i] = V[j] = V[k] = 96 \), then three bits are marked as errors. Now this error vector \( E[i] \) is added to \( S[i] \) since \( S \) is the unprocessed received vector whose errors are detected and marked in \( V \). This addition is modulo 2 which convert the erred bits back to the corrected bits. Note that the entire multiplication is nothing but a simple decoding process to find out the erred bits as any matrix multiplied to its transpose yields an identity matrix \( I \). And any other matrix multiplied with \( I \) yields the same matrix.

Figure 6.2 shows the garbled bits of cover media, a perfect solution for steganographic operation. The text area just below that is dedicated for writing secret messages to be sent with the option of encoding as many stega bits (between 1 and 3) per codeword as one wants with the facility of selection between Mode 1 and Mode 2. One such operation is shown in Figure 6.2. The message to be sent secretly is “It is a beautiful and sunny day, also not hot either. Hope it remains same throughout the day” with 2 stega bits per codeword. Mode 1 encoding is selected. The result of this operation is shown in the following picture. The genuine channel error probability induced is 0.01, i.e. every 100th bit is corrupted for large stream of binary code. With 2 stega bits per codeword and Mode 1 encoding, the following message is received at the receiving end after the decoding process is done on the received vector.

In Figure 6.3 (p. 35), six of the 93 bits have been changed such as 'throughout' became 'thrOughout' and 'beautiful' became 'bea'tiful'. Even though we have a very good decoding mechanism, these errors occurred as there must have been cases where there were more than 2 errors per codeword. This happened because of the genuine channel error which was introduced for every 100th bit. This channel error not only corrupted the respective bit,
but also carried the error throughout the communication which made the difference
eventually as it was unexpected at the receiver. Due to this nature, the decoder did not
attempt to look out for more than two errors in the received vector and the translation of
binary bits to text showed its impact in form of six corrupted bits.

### 6.3 MODE 2 DECODING WITH DETECTION RULE

In the case of Mode 2, the pseudo bits are inserted randomly by the encoder. Hence at
the receiver's end, to decode the received bit stream the decoding mechanism explained
above is not enough. The only knowledge the decoder has is the number of pseudo bits being
inserted per codeword. As explained in the previous chapter, to avoid the confusion of
numbering the decoded stega bits, the stega bits are inserted in equal length intervals at the sender.

The majority of the decoding rule used for Mode 1 remains same for the Mode 2 until we receive vector V i.e. the resultant matrix which has error indicators such as 120, 176 or 96. But since there are genuine channel errors as well artificially inserted random channel errors, the chances are very high that some of the stega bits are not being detected. Hence the detection rule is defined for Mode 2 decoding in such a way that the stega-channel is modeled as a binary erasure channel. In this case the performance may be greatly improved by using a code on this channel that is both capable of correcting and allowed to correct erasures and errors. Also as a consequence of the definition of a stega-channel as BSC, the steganographic information should be preprocessed; i.e. compressed, encrypted, and encoded using an error-correcting code the resulting stream is the sequence of bits to be sent.

The question arises of when to bring this detection rule into the picture. Let us suppose that 2 stega bits per codeword were encoded at the sender side and sent. At the receiver end, the decoder was able to decode both the stega bits and receive the message back. This was an ideal and perfect case. But suppose the decoder was able to decode just 1 stega bit or 0 in the worst case. This is where the detection rule is applied. The error matrix E is defined for each decoded codeword where entries of E, i.e. E[i] is 1 whenever an error is
detected and otherwise 0 whenever the normal bit is received. This error is nothing but a stega bit being decoded. The algorithm of detection rule is explained below.

- Check the number of stega bits S decoded.
- If S < Expected number of stega bits per codeword, then apply detection rule as follows:
  - Calculate the number of occurrences of 1 and 0 for the decoded codeword;
  - Save them as N(0) and N(1);
  - If N(0) < N(1) → set the stega bit as 0;
  - If N(0) > N(1) → set the stega bit as 1;
  - If N(0) = N(1) → make a random selection between 0 and 1.

Various examples are discussed. If \( V = 1111111111 \) then very obviously stega bit will be 1. Similarly if \( V = 0000000000 \) then stega bit is set as 0. When \( V \) is 10101101001110... which is the generally case, calculate N(0) and N(1) and apply the rule above. Use of detection rule does not guarantee the correct receipt of stega bits as the entire algorithm is probabilistic and based on assumptions. There is every possibility that the detection rule can go wrong at times, but by applying this rule it is made sure that a stega bit is set whenever it is not detected by the decoder. The code used to program this detection rule is shown below:

```java
if(numErrors == 2){
    int count = 0, e1 = 0, e2 = 0, onea=0, zeroa=0, oneb=0, zerob=0,
    for(int i = 0,i < 23returnedString.length,i++){
        if (E[i] == 1){ count++, }
    }
    if(count < numErrors){
        for(int l = 0,l<= 11, l++){if(E[l]) == 0){ e1++, } }
        for(int k = 12, k<23, k++){if(E[k]) == 0){ e2++, }  }
        if(e1 == 0){
            for(int l = 0,l <= 11, l++){
                if(returnedString[0][l]==0){zeroa++,
                if(returnedString[0][l]==1){onea++,}
                }
            }
            if(onea > zeroa){collectSecret[(2*i)+0] = 1,}
            else if(zeroa > onea){collectSecret[(2*i)+0] = 0,}
            else if(zeroa == onea){collectSecret[(2*i)+0] = r.nextInt(2),}
        }
        if(e2 == 0){
            for(int l = 12,l < 23, l++){
                if(returnedString[0][l]==0){zerob++,
                if(returnedString[0][l]==1){oneb++,}
            }
```
if(oneb > zerob) {collectSecret[(2*i)+1] = 1;}
else if(zerob > oneb) {collectSecret[(2*i)+1] = 0;}
else if(zerob == oneb) {collectSecret[(2*i)+1] = r.nextInt(2);}
}
}

The example considered here is for 2 stega bits per codeword. The detection rule is applied in two stages. For Mode 2 encoding, the random index is selected for different sets such as between 0-11 and 12-23 for 2 stega bits and 0-7, 8-15, and 16-23 for 3 stega bits. Hence at this decoder, the stega bits are also expected for each of these different sets. Hence the error matrix E is searched for entry 1 in sets of 0-11 and 12-23. If E is 1 then the counters e1 and e2 are increased by 1. If e1 > 0, then only the detection rule is applied for 0-11 set which calculates N(0) and N(1) (here zeroa and onea) for decoded string V (here returnedString[][][]) and the respective stega bit for that set is assigned according to the condition being raised for the comparison between zeroa and onea. Here stega bit is stored in collectSecret[] matrix.

Figure 6.4 shows the use of Mode 2 with 3 stega bits encoding per codeword. The message being passed is “Cricket is a very popular sport in indian sub-continent as well as many commonwealth countries.” After going through the decoder and ultimately by applying detection rule at stages, the message received is “Criaket is a v(ry popul2r sport in iwdian sud-contin]nt as wbll as mans comMonwEalth countr-,s” (see Figure 6.5, p. 39).

Twelve of the 95 bits were changed in the decoding process. As compared to Mode 1, the number of errors detected is almost the double (6 in 93 for Mode 1). But this behavior is expected as 3 stega bits are encoded per codeword which randomly inserts the stega bits at three positions and hence the decoder expects the stega bits getting decoded at three positions per codeword. If not then detection rule is applied, sometimes thrice per codeword which chooses some stega bit own its own.

Hence it proves that the random method of Mode 2 brings more erroneous bits at the receiver as compared to Mode 1 but it also proves that Mode 2 is more secure compared to Mode 1 as the codewords encoded in Mode 2 are not easily decodable.
The encoded text file is

Cricket is a very popular sport in Indian sub-continent as well as many Commonwealth count

Figure 6.4. Encoding of a secret message: Mode 2.
Figure 6.5. Decoding of a secret message: Mode 2.
CHAPTER 7

OBSERVATIONS AND CONCLUSION

As we know more about the different modes and their execution, the very question that comes to the mind is which mode is supposed to be used and where. The usability of these different modes varies from situation to situation and from scenario to scenario. Hence the basic few differences between Mode 1 and Mode 2 are covered and also examples of different scenarios are presented in order to get more insight about the usability standpoint of Mode 1 and Mode 2 below.

7.1 MODE 1 VS. MODE 2

The basic difference between Mode 1 and Mode 2 is that the encoder in case of Mode 1 encodes bits of cover media at fixed positions and in case of Mode 2 in random locations. The number of corrupted bits depends upon the filter chosen by the encoder but it should not exceed three as the decoder of Golay code is not capable of correcting more than 3 errors per codeword. Due to this property of Mode 1, it is more vulnerable to the third party attacks. For example, suppose an intruder manages to enter the steganographic system and he finds out the encoding mechanism, the intruder can easily catch the secret message being passed in the cover media text file as he has the knowledge of which positions per codeword are to look out for. Secret message bits can be easily captured as their position remains the same over all the 23 bit codeword blocks. For Mode 1 the decoding mechanism is also very simple at the decoder end. Hence from the usability perspective, Mode 1 techniques are utilized when the steganographic system to be configured needs to be fairly simple and is not supposed to have hefty overhead. Also since almost entire secret message is recovered at the receiver's end for Mode 1, it makes sense to utilize Mode 1 systems in places where message integrity is more important, i.e. places where receipt of the correct message is more important than the security of the message being transmitted.

For Mode 2 systems, they are very secure as far as eavesdropping is concerned. Even through intruder reaches to the small codewords of the secret message embedded cover
media, he is unsure about which bits to fetch for as the stega bits position is totally random per codeword. Even if he tries to use the trial and error mechanism, due to the property of random numbers, the probability of generating same set of 3 random number for 23 bits is \((1/23)^3\) which in mathematical terms turns out to be a fairly small number. For the decoder at the receiver's end, the life is simpler than the intruder since the decoder is aware of the decoding technique as well as the special detection rule designed for Mode 2. In spite of that, there is fewer guarantees as compared to Mode 1 that the secret message is entirely decoded and correctly received. Hence Mode 2 is used for applications where creating and maintaining a secure communication link is far more important than the integrity of the message, such as military where part of the receipt of the correct message can be traded off in order to achieve more secure communication which is difficult to decipher.

7.2 Observations

The steganographic software is run for variety of cases for both the modes with input being different set of cover media and secret messages. The outcomes and the secret messages decoded by the software at the receiver's end for these disparate set of input conditions have been noted down for n number of test cases and an average reading is calculated.

When the genuine channel error probability was tuned to 0.01, which is nothing but corrupting every 100th bit of input stream, observations were noted down per 100 bits received (see Table 7.1).

<table>
<thead>
<tr>
<th>Table 7.1. Corrupted Bits per Stega Bits Sent for Mode 1 vs. Mode 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of stega bits sent per codeword</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td><strong>Mode 1</strong></td>
</tr>
<tr>
<td>1 stega bit</td>
</tr>
<tr>
<td>2 stega bits</td>
</tr>
<tr>
<td>3 stega bits</td>
</tr>
<tr>
<td><strong>Mode 2</strong></td>
</tr>
<tr>
<td>1 stega bit</td>
</tr>
<tr>
<td>2 stega bits</td>
</tr>
<tr>
<td>3 stega bits</td>
</tr>
</tbody>
</table>
The results clearly show the considerable difference between the errors detected in Mode 1 and Mode 2. Also in both the modes, as more number of bits per codeword is encoded, the number of errors decreases at the receiver's end. The numbers remain fairly consistent for all the three cases in Mode 1, but for Mode 2 the difference between 1 stega bit sent per codeword and 2 stega bits sent per codeword are large. This phenomenon is observed due to the fact that the detection mechanism in Mode 2 becomes much more stringent and expectation of erred bits is more accurate as we increase the bit frequency at the sender.

7.3 Future Enhancements

The existing application is standalone i.e. a desktop application. The Java application runtime (.jar) that is created using the eclipse IDE which is nothing but an archive or just like a zipped package is an executable which can be moved to any workstation irrespective of the operating system that the workstation has and a single click can easily install the entire system on the respective machine.

The scope of the project can be extended by turning the software into a web application where the system needs to be loaded only on the machine containing the web server and can be accessed anywhere using remote IP address. Also the web support can be extended by adding an e-mailing feature to the software where the encoded stream can be mailed to the receiver to get it decoded. This feature can make the software really efficient.

Also the software is designed for cover media as a text file. In future, support for office word files or spreadsheets can be also included in the software by studying and analyzing the header of these file formats. The support for jpeg or gif images can also be added to the software where the image hides the secret steganographic message, which could be more concealing than hiding text into text. Other formats of cover media such as embedded audio or video files such as mpeg or avi can also be used.

Since the error correcting codes studied in detail before the software development started were Golay codes and hamming codes, Golay codes were chosen considering their stability and strong ability to detect and correct errors. There are equally powerful algorithms available such as BCH codes, cyclic codes which can detect and correct n number of errors.
into a codeword. The software can ask a user to select the encoding algorithm of his choice in between the algorithms that were just mentioned.

Finally the techniques that make any cryptographic algorithm stronger can be adopted in case of steganographic communication. An example of such techniques is the use of digital signatures. A digital signature is an addendum or a checksum which is appended at the end of the encoded stream and as a part of that stream over the communication channel. By adding a digital signature, the receiver can never deny or refute that the intended sender sent the encoded stream, thus by ensuring some level of message authentication. This authentication method provides a proof of integrity and origin of data.

These kind of small additions can be tried for the software to make it more interactive, user friendly and technically more sound and secure in combination of one or the other. A piece of web-based software that encodes the message and covers it using an image, has an e-mail feature, and ensures message authentication would really be compelling.

**7.4 Conclusion**

The steganographic communication is a fantastic concept of the class of information security. The two modes used with the encoding and decoding techniques of Golay codes solve the purpose of storing the secret message in a cover media work very nicely and effectively. The techniques discussed here to achieve such kind of a covert communication is just an effort to show that practically any kind of secret communication is possible in information security. There are few shortcomings but these can be overcome by enhancing the software just discussed before. All in all, this software was really interesting to work with, challenging and very informative as far as fields such as computer science and mathematics are concerned.
WORKS CONSULTED

