Information Hiding in DEFLATE Compressed Files

By

Sidra Khan

2008-NUST-MSCCS-12

Supervisor

Dr Fauzan Mirza

Department of Computing

A thesis submitted in partial fulfillment of the requirements for the degree of Masters in Computer and Communication Security (MS CCS)

In

School of Electrical Engineering and Computer Science,
National University of Sciences and Technology (NUST),
Islamabad, Pakistan.

(Feb 2012)

Approval

It is certified that contents and form of the thesis entitled **"Information Hiding in DEFLATE Compressed Files"** submitted by **Sidra Khan** have been found satisfactory for the requirement of the degree.

Advisor: **Dr Fauzan Mirza**

Signature:__________________

Date:__________________

Committee Member 1: **Dr Hafiz Farooq Ahmad**

Signature:__________________

Date:__________________

Committee Member 2: **Dr Zahid Anwar**

Signature:__________________

Date:__________________

Committee Member 3: **Mr Kamran H. Zaidi**

Signature:__________________

Date:__________________

Abstract

Steganography has been used as a way of secret communication such that no third party can suspect the presence of communication link. The media for steganographic communication kept changing on with the advancement in technology over the time. Compressed files and archives can also serve as a medium to carry hidden information. In this research, a popular compression algorithm DEFLATE has been studied and analyzed for information hiding purposes. DEFLATE is implemented with buffers and multiple flush modes have been provided to avoid buffer latency issues. In this study the flush operation during compression has been exploited to embed additional information inside a file and two schemes for information hiding in DEFLATE compressed files have been proposed. The proposed schemes embed additional information during the compression process of DEFLATE and produce a steganographic compressed cover file with additional information embedded inside it. The proposed information hiding schemes are secure and provide good information hiding capacity. A ratio of size of the compressed file and embeddable data size has been calculated and a threshold value is defined. The proposed scheme works well with single file compression and information hiding but can be adapted and implemented to communicate secret information over the network using protocols that support DEFLATE compression.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person nor material which to a substantial extent has been accepted for the award of any degree or diploma at SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at SEECS or elsewhere, is explicitly acknowledged in the thesis.
I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Sidra Khan**

Signature: __________________

Acknowledgments

I would like to express my deepest gratitude to Allah Almighty who enabled me to accomplish this hectic challenge in sound health, to my parents for their support, and motivation. I am heartily thankful to my supervisor Dr Fauzan Mirza for his support and guidance throughout the research period. Lastly, I offer my regards and blessings to all those who supported me in any respect during the research. May Allah bless all of us! Amin

Sidra Khan

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

1.1 Introduction

Steganography is not a new term; it has been used over the ages by both criminals and law enforcement people for secret communication. It is defined as follows:

> "Steganography is the art and science of writing hidden messages in such a way that no one, apart from the sender and intended recipient, suspects the existence of the message, a form of security through obscurity. The word steganography is of Greek origin and means concealed writing. " [1].

Steganographic communication model consists of a sender, Carrier medium, a receiver and secret data to communicate. Steganography is all about hiding the presence hidden message in way that any third party cannot suspect the presence of secret data communication. Old methods used for steganographic communication were invisible inks, character arrangements, tattooing in the scalp of slaves, and microdots etc. In recent times steganographic communication has been done by using covert communication channels in which data is hidden in some type of binary files also known as cover files or carrier medium. After data embedding it is modified to steganographic medium. Presently, most commonly used steganographic media include images, audio, video, executable files, compressed files etc. In an effective steganographic communication sender and receiver has to agree upon the carrier medium i.e. set of files to transmit hidden data and the steganographic software usage. To enhance the security level mostly the secret data is encrypted with special passwords so that the data remain unusable if the presence of data is detected by forensics examiner or any third party.

1.2 Evolution of Steganography

Over the history people have been using variant methods to hide information e.g. Greeks used to write message on wax-covered Tablets and made it appear blank to hide its presence, another method was to tattoo a message or image on the shaved head of messenger. The message would remain undetected until the head was shaved again to reveal it. In early World War II most of the secret communication was done through invisible inks. Null cipher (unencrypted messages) was also a famous way to hide information within text. In this method the actual message is concealed within an innocent message to deceive the observer. Microdots technology was invented by Germans to communicate large amount of data including photographs and drawings. Microdots were microfilm chips created at high magnification of the printed period size with the clarity of standard sized type written pages. As the technology grew with time old information hiding methods were given new twists for better obscurity. Presently the focus of steganography has been shifted to digital technologies such as images, text files, audio files, video files etc. Hidden messages can be embedded in images or in audio portions of the broadcasts to be communicated safely. There are a lot of techniques or algorithms to hide messages inside digital media most probably used for confidential information exchange for both legal and illegal purposes. Software watermarking is also an important application of steganography. Information in the form of Watermark is embedded inside the software to impose the intellectual property rights of the owner of specific software and to detect piracy. Another form of watermarking namely Image watermarking can be very helpful in detecting image tampering especially in Medical images. Hence steganography has been used for both malicious and legitimate purposes throughout the time.
There are a lot of tools and software developed up till today for hiding information within different media. The strength of a steganographic tool depends mainly on making information more and more obscure to make it undetectable to steganalysis i.e. analysing file for hidden information. The steganalysis technique is applied on file having hidden information as per nature of information embedding method. The strength of a steganographic approach is measured by its resistance to steganalysis.

1.3 Steganography in Compressed Files

Apart from traditional ways of steganography modern steganography paradigm has been shifted to new approaches such as hiding information in compressed files i.e. Zip, cab, GZip etc , and in file systems i.e. Stego File systems. Compressed archives are a great source to hide large amounts of data having the advantage that existing methods cannot do well to detect the presence of hidden data in compressed files [3]. There are a lot of data hiding schemes but most of the existing ones are subjected to limitations in data hiding capacity, security, and robustness.

1.4 Motivation

A variety of Information hiding methods for compressed files exist but most of the methods are based on altering the values of different header fields present in file format specification and provide limited data hiding capability. Additionally the changes made by these methods to compressed files can corrupt those and are easily detectable if file is analyzed in detail with certain hex editing tools. Hence there is need of an information hiding method which cannot change or damage the compressed file format, is secure, can hide large amount of data without increasing size of compressed file and do not degrades compression speed. Therefore, the research problem comprises of analysis of Deflate compression algorithm for finding additional ways of information hiding in compressed files and to devise a reversible information hiding scheme for deflate compressed files.

1.5 Challenges and Goals of Research

The challenges and aims of this research are explained in the following:

- To devise lossless secure information Hiding scheme i.e. only the intended recipient can recover hidden data from the file.
- To devise a scheme which does not change or damage the archive file format i.e. to keep hidden data undetectable.
- To ensure that the size of a compressed file with hidden data should not be greater than the original file.
- To ensure that the process of information hiding does not degrade the compression rate.

Chapter 2

Literature Review

This chapter discusses the evolution of steganography in compressed files. Information hiding schemes for compressed files developed up till now are mentioned and are critically analyzed.

2.1 Evolution of Steganography in Compressed Files

File compression is a widely used process to reduce the size of a file for the purpose of fast and efficient communication. There are many file compression tools available employing variety of file compression algorithms. These compression algorithms are applied according to the nature of data present inside file i.e. certain algorithms can do well with image files while text, audio and video files can be better compressed by different techniques. Many compressed file formats have been designed up till now to represent compressed data. Among those most commonly used formats are Zip, RAR, Cab, GZip etc. These file formats are used for file compression and decompression as well as archiving i.e. combining multiple files into a single file using compression.

Information hiding methods proposed up till now for compressed files fall into two categories. Few of the methods hide data in compressed files by exploiting the structure of file format i.e. manipulating the values of various fields supported by file format while others exploit compression technique to hide user intended data into file during the process of compression. This hidden data is extracted from file during or before decompression of the file. Steganography in zipped archives is achieved by exploiting zip file structure [3]. In zip file structure different fields are used to store metadata in header of the file the value of these certain header fields can be modified to conceal the

presence of hidden data e.g. utilization of "EXTRA field" to store data file to hide its presence from archivers but present in archive and extractable when needed [3]. The purpose of EXTRA field is to store additional information about file such as encryption information etc. but it is rarely used. Other ways used to hide data in archives by exploiting header structure includes changing the value of central directory pointer to user given position to hide the presence of certain files. Every zipped file inside a zip file has an entry to the central directory but another way to hide file is to write zip entries of files without adding them to central directory [4]. Files hidden using these methods are usually protected by self destruction mechanisms or by adding certain malwares in archives so that hidden files strip off by antivirus program if captured in wrong hands [3].
Popular data compression algorithms include Huffman coding, LZ77, LZW, DEFLATE etc There are methods which exploit the working of a compression algorithm to hide additional data inside compressed file E.g. in Huffman coding a Huffman tree is constructed representing the compressed code for each symbol present in data. To hide additional data during the process of tree construction each symbol is encoded with an additional secret bit of data to be hidden. The compressed code constructed in this way has one secret bit hidden for each symbol encoded during compression [5].

2.2 Overview of Existing Techniques

Few methods of information hiding in compressed files proposed up till now are discussed in the following.
In 2006 K yoshioka, K Sonoda, and O Takizawa presented a new method for information hiding in compressed data. The proposed method hides certain information in the compressed file in a way that it can still be decompressed in the same way as a compressed file without hidden data is decompressed. The secret key used to embed data can also be used to extract hidden data from the compressed file. There proposed scheme works on the principal of lowering compression rate in order to increase information hiding capacity of the compressed file. The chosen compression scheme is LZSS a variant of LZ77 algorithm. Since LZSS is a dictionary based algorithm so it compresses data by finding correlations and replacing these with their previous references. The data to be hidden is embedded by finding multiple matches or correlations in dictionary and leverage them to hide information. In order to increase information hiding capacity the authors have modified LZSS compression i.e. searching for shorter matches instead of long matches during LZSS encoding to hide extra data. They also defined a threshold parameter

to obtain a balance between embeddable data size and compression rate [12]. In 2009 another information hiding scheme for Huffman coding compressed files was introduced by K N Chen, C.F. Lee, C. C. Chang, and H-C Lin. This scheme works by exploiting the working of the Huffman coding scheme. It was introduced to hide information e.g. image, doc, video in Text files. As Huffman compression algorithm works by constructing a Huffman tree containing variable length code for each symbol. During the encoding of symbols each symbol carries one extra secret bit of information which contains the hidden data. So each symbol carries one bit of information with it. During the decoding process these secret bits are extracted from the symbols [5].
In 2010 Mario Vuksan, T Percin, Brian Karney published a detailed study of popular archive formats including Zip, Gzip, CAB, and 7-Zip exploring maximum possible options of hiding data inside and programmed a tool named"NyxEngine" to analyze an archive for the hidden data. As per their study the schemes for hiding information in archives usually exploit the file format specifications e.g. using rarely used file fields to hide data. So for analysing an archive for compressed data the inspection test should identify the file format, validate it to know that any field is not misused, and check it byte by byte to detect the presence of any hidden information. Information can be hidden in a Zip file by different means e.g. by modifying compressed file name, using file comment field, utilization of Extra field, making changes to internal structure of a zip file, Hiding the presence of a specific file in an archive, and injecting secret data in a file. "Zipped steganography" by Corinna John [4] and "ZJ Mask" by Vincent Chu [14] are two famous steganography tools for zip files. In case of RAR files information can be hidden by modifying header flags i.e. By applying password for first file only. In CAB File format the "decompressed size" field can be modified to make file an archive bomb. In GZip documented extra fields can be added to store and hide information. The 7-zip archive format can be exploited by tampering header CRC and modifying other header fields etc. The "NyxEngine" inspects a compressed archive for hidden data. It processes an archive analyze it for the presence of hidden data. It is capable of recovering broken or hidden files. NyxEngine starts working by identifying archive format, then browse it for the packed content, validate the format of archive to detect any tampering and search steganographic information from it [3].

2.3 Critical Review

All of the existing techniques for hiding information in compressed files or compressed archives have certain limitations in term of Information hiding capacity and methods. Most of the schemes work by exploiting file format structure which can damage the archive file and make it vulnerable to doubt the presence of hidden data in it. The Extra hidden information can sometimes raise the file size to a bigger number. A ratio between compressed file size and information hiding capacity has to be measured to devise a safe information hiding scheme. The information hiding method should ensure the safe recovery of hidden from the compressed file.

2.4 Problem Description

Steganography provides a way of communication in which two parties can secretly communicate data in such a way that any third party cannot detect or interpret the communication. Compressed files can act as a medium to hide information inside the file in way that the compressed file appears like normal benign compressed file in structure to the third party viewer but it contains hidden data which can be detected and interpreted by the intended recipient. The medium to embed data namely cover file can be an image, audio, video or text file. A good information hiding scheme should be lossless, secure, and efficient in terms of compression ratio and speed.

In this research a new method is proposed for hiding information in compressed files. For this purpose the "DEFLATE compression Algorithm" is analyzed. During study of DEFLATE compression algorithm certain vulnerabilities are discovered which can be exploited to develop a novel information hiding scheme for text files. The working of DEFLATE compression algorithm is discussed in chapter 3. Chapter 4 describes the proposed steganographic method.

Chapter 3

Introduction to DEFLATE

The compression scheme studied and analyzed in this study is DEFLATE compression algorithm. This chapter gives a detail introduction to DEFLATE compression and working of a freely available source code library for deflate namely Zlib.

3.1 What is DEFLATE?

Deflate is a popular lossless data compression algorithm. It has been documented in RFC 1951 but was designed earlier by Phil Katz for his PkZip archiving tool. It is supported by most of the compression utilities as compression scheme for file compression as well as compressed data transmission over internet e.g. HTTP. Deflate compression scheme is the basic method used in ZIP, GZip, PNG.
Deflate compression algorithm is a combination of LZ77 coding and Huffman coding applied in arbitrary order. Deflate process data bytes and the coded data consist of blocks of arbitrary sizes including both compressed and uncompressed blocks. To understand Deflate compression process completely one must have an introduction to the working of Huffman and LZ77 coding.

3.1.1 Huffman Coding

It is a lossless data compression method also referred as entropy encoding algorithm presented by David Huffman in 1952. Huffman coding process produces variable length codes for each unique symbol from the source file depending upon its frequency of occurring in the input file. Huffman scheme works well in scenario when the data set to be compressed is produced in advance because it reads the source file twice, firstly to calculate symbol

Symbol	Frequency	Code	Code length	Total length
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

Table 3.1: Huffman Codes for symbols [6]

frequencies, secondly to assign compressed codes to the data in source file. Huffman method is preferred when the input file has characters with random probability otherwise for uniform probability data this scheme cannot perform well.
Huffman compression works on the basis of idea that more frequently occurring characters in an input file can be encoded with fewer bit codes. The compression process initially begins with reading the source file and calculating frequency or probability values for each unique symbol present in the file. In next step the two symbols with the lowest frequencies are combined to form a binary Huffman tree with a parent node having frequency equal to sum of the frequencies of the leaf symbols acting as leaves. The parent node also becomes a part of the set of symbols with frequencies. Then further two symbols with lowest frequency in the set are selected to join the Huffman tree. In this way all of the symbols in the set are joined to form a complete binary Huffman tree. During the tree construction process the variable length codes for each symbol are also assigned. The code for a symbol can be calculated by traversing the Huffman tree from top to bottom in a way that every left child node is given a zero value while every right node is given a one value. An example of Huffman coding is discussed in the following. In table 3.1 there is a List of symbols with respective frequencies in increasing order.The Huffman tree for the above symbols can be constructed as shown in figure 3.1.Huffman codes assigned to symbols are given in table 3.1.[6].

Symbol	A	B	C	D	E
Frequency	24	12	10	8	8

Total bits transmitted using Huffman encoding are 138 for the above frequency distribution if static codes had been used then total bit length would be larger for the data. Huffman decoding can be done by reading the Huffman tree from top to bottom as per input stream i.e. for 0 move to

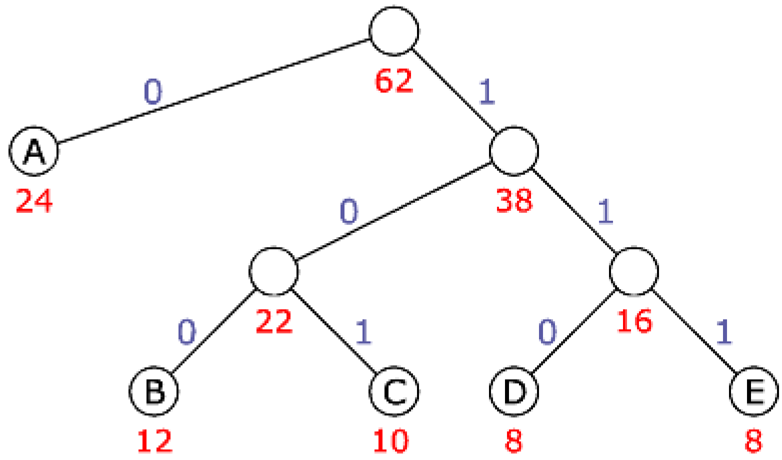

Figure 3.1: Huffman code tree [6]

left side and for 1 move to right side of tree when a leaf is encountered one character has been decoded. Hence frequently occurring symbols get shorter code and less frequently occurring symbols will get larger code lengths. For one set of data symbols multiple Huffman trees can be constructed with slight difference. So, for deflate compression two simple rules has to be followed in constructing Huffman tree:

1. Elements having shorter codes are placed left of the elements with longer code.

2. If multiple symbols have same code then symbols appearing first in the data set would be stored left of the tree.

By applying these rules only one Huffman tree is possible for one set of symbols and for the process of decompression these code lengths have to be transmitted to reconstruct the tree for decoding of data [7].
This is a very simple example of Huffman coding process. It can be applied to large data files in same way. Many variants of Huffman coding are also available including n-array Huffman coding, Adaptive Huffman coding etc. Adaptive Huffman coding is an advanced form Huffman coding based on the same principle with a little variation. Since for Huffman coding or static Huffman coding the compressor should know complete data to be compressed in advance to calculate exact frequencies of the symbols so it is not applicable to scenarios where data is produced at random intervals and compressed simultaneously. Adaptive Huffman coding determine symbol frequencies dynamically as input stream is produced and updates the Huffman tree struc-

ture accordingly to match the new values. In deflate both static and adaptive or dynamic Huffman coding is used as per requirement. Variants of Huffman coding are widely used in many applications to compress and transmit data simultaneously.

3.1.2 LZ77 Coding

LZ77 is a lossless data compression algorithm published by Abraham Lempel in 1977. It provides the base for many advanced compression algorithms such as LZW, LZSS, and LZMA etc. LZ77 compression method comes under the umbrella of dictionary coding schemes. Dictionary based compression schemes works by finding correlations or similar patterns in the data file and replacing there further occurrences with reference to a dictionary. Dictionary can be created statically or adaptively. In LZ77 dictionary creation is adaptive i.e. No need to transmit dictionary to receiver but can be created at run time.
LZ77 compression is performed by finding repeated occurrences of data and replacing it with reference to its early occurrence in the input file. A match found is replaced with length-distance pair value. Length parameter refers to the length of the matching pattern while distance or sometimes referred as offset value tells the distance of the pattern from its previous occurrence in the data. To find out a match further the encoder has to keep track of the previously read or recent data. For this purpose a buffer or a sliding window of data is maintained that is why LZ77 is also referred as sliding window compression. This sliding window or buffer has specific size value e.g. 4KB, for DEFLATE 32KB window etc. A match is found within that specific range of data. Encoder and decoder both have to maintain this sliding window. Here is an example to illustrate the LZ77 compression more clearly.

Data: abcdefghijAabcdefBCDdefEFG
Output: abcdefghijA {distance: 11, Length: 6} BCD {distance: 6, length:3} EFG

In above example the lZ77 encoder replaces the repeating pattern with distance value to its previous occurrence and length of the repeating pattern. On the decoder side same size sliding window is maintained which replace the length distance value of a match with its original value through moving backward by a length of the distance value. In the same way LZ77 compresses the data stream in DEFLATE compression used with Huffman compression arbitrarily.

3.2 DEFLATE: How it Works?

Data compressed by DEFLATE consist of series of blocks of arbitrary sizes except uncompressed blocks which cannot exceed the size limit of 65,535 bytes. Each block is compressed by both LZ77 coding and Huffman coding. The Huffman tree is different for each block and is not dependent on the previous ones but in LZ77 coding matching patterns can exist within the previous 32K values that may include previous blocks as well. Every compressed data block consists of two parts mainly:

1. Compressed Huffman code trees (compressed by Huffman encoding) representing compressed data.
2. Compressed data.

Since in LZ77 compression only the matches are replaced by references to their previous occurrences and other characters are plainly written in the output stream. Therefore, the DEFLATE compressed stream has three entities:

1. Distances or offsets of previous matches
2. Length of the match
3. Literals or uncompressed characters

The distance or offset value can be up to 32K and length value can be of 258 bytes. Separate Huffman code trees are constructed for distance, length and literal values and are assigned variable codes from different code tables. These code trees are present before the respective data blocks [8].

3.2.1 Deflate Block Format

Deflate compressed data blocks have certain format. Each block has a header and a data part. Header precedes the data part. Starting three bits of the each block are header bits. In which first bit is “BFINAL” and last two bits are of “BTYPE” i.e. Block type. BFINAL bit value is 1 for last block of data otherwise 0. While BTYPE value as per [8]reveals how data is compressed as described below:
BTYPE 00 – No compression
BTYPE 01 – Compressed with fixed Huffman codes
BTYPE 10 – Compressed with dynamic Huffman code
BTYPE 11– Undefined (error)
The Huffman codes for lengths/literals or distance values differ for two formats of data compression. The format of different blocks type is as follows:

Code	Extra Bits	Length(s)	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Figure 3.2: Huffman code values for Length parameter in Deflate [8]

Uncompressed Blocks (BTYPE 00)

In the uncompressed blocks of BTYPE 00, length of the block with its one's compliment is stored in start and after it the data part is stored.

Compressed Blocks

As mentioned above deflate encoded data comprises of three distinct entities i.e. literal bytes from the alphabet set of 0 to 255, or length-distance pair values where length can be any value between 3 to 258 and distance can by any value drawn from 1 to 32768 values. The literal and length values are merged into a single alphabet range 0 to 285 in which first 0 to 255 indicate literal bytes, 256 symbols is of End of block value, and 257 to 285 are for length codes as shown in figure 3.2.

- Fix Huffman codes Compression (BTYPE 01)

For fixed or static Huffman codes the literal/length alphabet codes are given in figure 3.4. The code bits given in figure 3.4 are sufficient for code construction for length and literal bytes while distance values are represented by 5 bit fixed codes ranging from 0 to 31 in figure 3.3. Literal/length value 286-287 and distance values 30-31 does not exist in data but take part in code construction.

Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Figure 3.3: Fix Huffman code values for Distance parameter in Deflate [8]

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Figure 3.4: Fix Huffman codes for Literal/ length values, [8]

```
0 - 15: Represent code lengths of 0 - 15
    16: Copy the previous code length 3 - 6 times.
        The next 2 bits indicate repeat length
              (0 = 3, ... , 3 = 6)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    17: Repeat a code length of 0 for 3 - 10 times.
        (3 bits of length)
    18: Repeat a code length of 0 for 11 - 138 times
        (7 bits of length)
```

Figure 3.5: Alphabet for code of Dynamic Huffman code compression [8]

- Dynamic Huffman code Compression (BTYPE 10)

In Dynamic Huffman coding the literal/length code and distance code occurs right after the header bits the code is defined by sequence of code lengths and the code lengths are again compressed with Huffman coding for more compactness. The alphabet for code sequence is shown in figure 3.5.The format for the dynamic Huffman compressed block is represented in figure 3.6.

3.3 Details of Compression Algorithm

During DEFLATE compression process the compressor can decide to terminate the currently processing block and to start a new one when compressor buffer fills or when it is useful. To find duplicated strings or repeated sequences the compressor uses hash chains. The algorithm discards old matches from hash chains at regular intervals to avoid long matches and for greater performance otherwise for improved compression the compressor searches for longer match even after one match is found it is known as lazy matching.

```
5 Bits: HLIT, # of Literal/Length codes - 257 (257 - 286)
5 Bits: HDIST, # of Distance codes - 1        (1 - 32)
4 Bits: HCLEN, # of Code Length codes - 4     (4 - 19)

(HCLEN + 4) x 3 bits: code lengths for the code length
   alphabet given just above, in the order: 16, 17, 18,
   0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

   These code lengths are interpreted as 3-bit integers
   (0-7); as above, a code length of 0 means the
   corresponding symbol (literal/length or distance code
   length) is not used.

HLIT + 257 code lengths for the literal/length alphabet,
   encoded using the code length Huffman code

HDIST + 1 code lengths for the distance alphabet,
   encoded using the code length Huffman code

The actual compressed data of the block,
   encoded using the literal/length and distance Huffman
   codes

The literal/length symbol 256 (end of data),
   encoded using the literal/length Huffman code
```

Figure 3.6: Format for the dynamic Huffman compressed block [8]

In normal situation the compressor search for a longer match after finding a long match for better performance but if compression ratio is considered then compressor keeps on finding longer matches. In Deflate implementation parameters can be adjusted to support or avoid lazy string matching [8]

3.4 Introduction to Zlib

Zlib is a general purpose library for data compression written by jean Loup Gaily and Mark Adler. It is an abstraction of DEFLATE compression algorithm used in GZip program. Zlib library is a by default component of many software platforms e.g. Linux, Mac OS X, and the iOS. This library was released in 1995 for public use till now many updated versions have been released. The latest version of Zlib library only support DEFLATE as compression algorithm though Zlib format header provides flexibility to add other algorithms. Zlib library compressed data is enclosed by Zlib or GZip wrapper to add Error correction and stream identification features which are not provided by raw deflate compressed data.
To get source code for DEFLATE compression algorithm Zlib library is a good resource as it provides the programmer with facility of flexible controls in terms of adjusting DEFLATE compression parameters as per user requirement. The parameters which can be adjusted include memory usage, control of processor, compression level to maintain a balanced trade off between compression ratio and speed, and optimized compression type for specific type of data[9].

3.4.1 Zlib Stream Data Format

A file compressed with DEFLATE using a Zlib wrapper have the following data format. The compressed stream starts with Zlib header. Its first byte indicates the compression method and flags (CMF). In CMF first four bits select compression method (CM) i.e. for DEFLATE compression CM=8, the next four bits tell compression info i.e. CINFO for CM=8 the value of CINFO is base 2 log of LZ77 window. The next byte in Zlib stream is flags byte (FLG). The first four bits of FLG are dependent on the CMF and FLG in a way that for the combine value of CMF and FLG the value of the following expression i.e. CMF*356+FLG is a multiple of 31. The next bit of FLG is FDICT. If FDICT bit is on it means that a dictionary identifier (namely DICT i.e. ADLER-32 checksum of dictionary byte) is present after the FLG byte. The last two bits of FLG reveal the value of FLEVEL which is compression level.[9] For deflate compression method the compression level

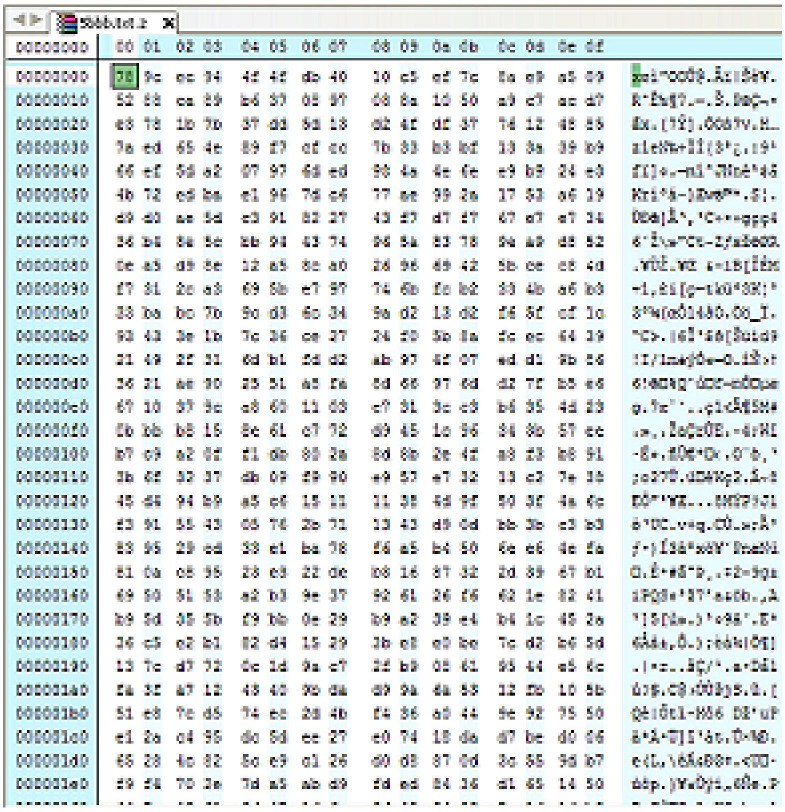

Figure 3.7: Hex dump of a DEFLLATE compressed text file

values can be as follows:
0 - Fastest algorithm is used by compressor
1 - Fast algorithm is used by compressor
2 - Default algorithm is used by compressor
3 - Max. compression is used by compressor, Slowest Algorithm
The format of the compressed data is according to the compression method i.e. DEFLATE compressed format. When decompresser starts decompressing a file it first ensures the values of the fields above explained to ensure the integrity and correctness of a compressed file [9].

3.4.2 Implementation Details of DEFLATE in Zlib

The Zlib library provides in memory compression/Decompression functions using DEFLATE Compression scheme. The compression can be performed in single step or it can be performed in multiple steps using repeated calls

of the compression function. It depends mainly on the size of buffer used for compression. For large buffer compression can be done in one step but for small buffers repeated calls of the compression function are necessary. The decompresser examines the integrity of data before decompression to ensure that data is not corrupted. By default compressed data format used is Zlib but optionally files can be read and written as GZip streams also.
Zlib implements DEFLATE compression with different functions. Two main functions include deflate() for compression and inflate() for decompressing data. The initial parameter values are initialized using deflateinit() and inflateinit() functions for compression and decompression e.g. Compression level, Flush mode to be used during compression if needed etc. As per [15]Zlib implements DEFLATE with different flush modes given in the following:
#define Z_NO_FLUSH 0
#define Z_PARTIAL_FLUSH 1
#define Z_SYNC_FLUSH 2
#define Z_FULL_FLUSH 3
#define Z_FINISH 4
#define Z_BLOCK 5
#define Z_TREES 6
Deflate() reads data from input file into a buffer and starts compressing it until the output buffer is filled up with data or the input buffer becomes empty. It can create "output latency" in way that the compressor is reading data but not producing any output.It can perform one or may be both of the following functions:

- It can keep on compressing input data available in buffer and constantly updating and telling the value of the remaining data available in the buffer to avoid a situation in which output buffer is already fill and no room for the newly produced output.
- It produces more output and keeps on updating the available output buffer space value. It can be done only if the value of flush parameter is non zero.

Flushing process is used less frequently and is preferred in web applications only. It is mentioned above that DEFLATE compression algorithm can be used with few streamed transport protocols e.g. TLS, SSH, PPP. These protocols transmit data by dividing data into packets. By using DEFLATE compression scheme these successive packets appear to be a part of single continuous compressed data stream and since DEFLATE works by using buffers so some kind of flushing is needed when transmitting data to ensure that every byte is received to peer without knowledge of any next data byte. Flushing is used for this purpose [10].

3.4.3 Zlib Flush Modes

Zlib implements DEFLATE with flush operations to avoid buffer latency. It provides four different type of flushing parameters described below:

Z_NO_FLUSH: It allows Zlib to accumulate large amount of input data in the input buffer for compression. The compression ratio is maximum with Z_NO_FLUSH mode.

Z_PARTIAL_FLUSH: It is the standard flush method in SSH protocol. A partial flush when applied, It processes any uncompressed input data present in input buffer in one or several byte blocks, sends an empty type 1 block of data, possible sends another empty type 1 (BTYPE 01) block.

Z_SYNC_FLUSH: It is most commonly used flush method in Zlib. A sync flush processes any input data which is not compressed into one or several blocks depending upon the size and nature of data and send an empty type 0 (BTYPE 00) block. An empty type 0 block may have 3-bits block header, 0 to 7 bits equal to 0 for byte alignment and the four byte seq. 00 00 FF FF.

Z_FULL_FLUSH: A full flush is rarely applied as it can degrade compression ratio badly. It is a variant form of sync flush. During LZ77 compression in DEFLATE a dictionary is maintained containing previous 32KB of data for finding correlations. This dictionary keeps on updating by deleting old entries and adding new ones. A full flush once applied can empty the complete dictionary. Since it is a variant of sync flush, it also includes 00 00 FF FF, and restores byte alignment. The decompression can be started from a full flush point without any knowledge of the previous bytes.

Z_BLOCK: Any data present in input buffer is compressed and emitted like sync flush but output not aligned on byte boundary. Up to 7 bits of the current block can be added to next byte after the completion of next deflate block. The compressor does not have complete data to decompress and has to wait for the next block to be emitted. Z_BLOCK is mostly used in latest web applications which works by controlling deflate block emission.

Z_TREES: It works same as Z_BLOCK flush mode. The Z_TRRES flush mode returns after end of header of each DEFLATE block to record the header length for later use in random access within a DEFLATE block.

3.5 Zlib Compression and Decompression

Zlib implements DEFLATE compression and decompression processes using different functions. The function provided for compression is deflate() and for decompression is inflate(). The compression process starts by initializing deflate compression parameters e.g. compression level, strategy, flush modes etc by calling deflateinit(). After initialization deflate () is called for compression. Deflate compression is a buffered process. Both input and output data is read into separate buffers of specific sizes. The buffer size is defined in advance before initialization of compression parameters. Zlib provides special return codes for compression and decompression functions. There positive values indicate normal events and negative values indicate errors given below as per [15]:
#define Z_OK 0
#define Z_STREAM_END 1
#define Z_NEED_DICT 2
#define Z_ERRNO (-1)
#define Z_STREAM_ERROR (-2)
#define Z_DATA_ERROR (-3)
#define Z_MEM_ERROR (-4)
#define Z_BUF_ERROR (-5)
#define Z_VERSION_ERROR (-6)
In the start of compression process data is read into input buffer from source file. Input buffer data is compressed until output buffer is not full. Data from source file is compressed until end of file. There can be a point when compression stops because input buffer becomes empty and output buffer becomes full. To avoid buffer latency flush is performed here e.g. Z_SYNC_FLUSH, Z_PARTIAL_FLUSH etc. Flush mode can be set for both compression and decompression functions. For successful compression of deflate() it returns Z_OK and if all input has been consumed it returns Z_STREAM_END. At the end of compression process deflateEnd() is called to free the allocated space to data structures. Zlib provides special function to adjust the deflate compression parameters e.g. compression level and strategy during compression namely deflatParams(). deflatesetdictionary() can be used to set the dictionary before the compression process. Dictionary is useful when

data is short and predictable. Zlib supports gzip compression format as well. Special functions are provided to write gzip header with compressed data.
DEFLATE decompression process starts with inflateInit() which initializes data structures for the decompression process. After initialization inflate() is called which reads the compressed file into input buffer and decompresses it. Flush mode can also be set in decompression process to avoid buffer latency. Return codes for decompression functions are also provided to indicate the return status of the processes. InflateEnd() is used to free the allocated space to data structures during decompression [15]. Zlib supports two check-sum functions to maintain security and consistency of the compressed data including crc32 and Adler32 checksum in which Adler32 checksum is much faster.

Chapter 4

Proposed Methodology

After studying DEFLATE compression algorithm few of the vulnerabilities have been discovered. The discovered vulnerabilities are exploited to devise two new schemes for information hiding in deflate compressed files. This chapter discusses DEFLATE vulnerabilities and the proposed information hiding schemes.

4.1 Vulnerabilities discovered in DEFLATE

Deflate compresses data files with a combination of dictionary based coding i.e. LZ77 coding and entropy coding i.e. Huffman coding. At the start of compression it reads data from the input file into a specific size buffer and then compresses this data. For large sized buffers the complete data file can be compressed in a single step but for small sized buffers the compression step repeats as long as complete input file is processed. The compression process of DEFLATE has certain vulnerable features which can be exploited in different ways to hide information. These vulnerable features are discussed in the following:

4.1.1 DEFLATE Buffer Latency Exploitation

It is mentioned above that DEFLATE reads input data in a buffer, compresses it, write to output buffer and then to the output file. During compression it is a possibility that input buffer becomes empty or the output buffer fills up leaving no space for writing output until the output buffer becomes empty. It can cause buffer latency i.e. reading input without producing output. This kind of situation can be avoided by using flush operations provided by the Zlib implementation of DEFLATE. Chapter 3 dis-

cusses the six flush functions provided by Zlib in detail. Most commonly used flush operations are Z_SYNC_FLUSH, Z_PARTIAL_FLUSH, and Z_FULL_FLUSH. These flush operations can be exploited to hide information inside a compressed file. By analysing Zlib working it is revealed that flush operations are called repeatedly during the compression of a file. These flush operations add few extra bytes in the compressed stream on every call e.g. Z_SYNC_FLUSH adds an empty type 00 block containing 00 00 FF FF value i.e. an empty type zero block.
According to the proposed exploit flushing is forced to perform and during flush operation e.g. Z_SYNC_FLUSH call instead of adding an empty block a block of secret information can be added here. A type zero block starts with a 3-bit header followed by length of the block with its one's complement, and then the data. A secret data block can also be added at the place where Z_PARTIAL_FLUSH is performed. Z_FULL_FLUSH is a variant of Z_SYNC_FLUSH. It also adds four byte sequence (00 00 FF FF) to the data plus empties the dictionary maintained for LZ77 matching. It is used to avoid long matches or lazy matches. It degrades compression rate but increases compression speed. A block of secret information can be hidden at the place where full flush is applied in the same way as it stored with Z_SYNC_FLUSH.
At the start of compression the user has to specify which flush mode to be used during compression. Number of flush operations performed during compression varies with different buffer sizes. A flush performed repeatedly will embed more data inside the compressed file.

4.1.2 Adjusting DEFLATE Parameter Values

According to Zlib 1.2.5 documentation [15]the values of certain DEFLATE parameters can be adjusted to user specified values as per requirement during compression. Hence during information hiding process in case flush operations compromise compression ratio these parameter values can be changed to improve the compression ratio.

1. **Memory Level:** This parameter specifies that how much internal memory has to be used by internal compression state during compression process. Memory Level: 1 uses minimum memory but reduces compression rate while memory level 8 while memory level: 9 is for maximum speed. The default value is 8.

2. **Compression Strategy:** During compression process the compression strategy is selected as per nature of the data e.g. the data produced by filter or predictor can be best compressed with Huffman coding.

Since user is least concerned about the compression strategy selected by compressor. This option can be manipulated as per requirement of information hiding. The available compression strategy options are given below [15]:
0 #define Z_DEFAULT_STRATEGY: used for normal data
1 #define Z_FILTERED: For data produced by filter or predictor (More Huffman & less string matching).
2#define Z_HUFFMAN_ONLY: To force Huffman coding only no string matching
3#define Z_RLE: To limit match distances to 1 (Run length encoding better compression for PNG data).
4 #define Z_FIXED: use of fixed Huffman codes.

3. **Compression Level:** It can also be adjusted as per requirement [15].
#define Z_NO_COMPRESSION 0
#define Z_BEST_SPEED 1
#define Z_BEST_COMPRESSION 9
#define Z_DEFAULT_COMPRESSION (-1)
Z_NO_COMPRESSION does not compress data at all and simply writes the input data to output file in blocks. Level 1 can be used in scenario where compression speed is important while level 9 provides the highest compression ratio. Z_DEFAULT_COMPRESSION is used mostly as it provides a default balance between compression ratio and speed.

4.2 Proposed Steganographic Schemes

Based on the above discovered vulnerabilities two steganographic schemes have been developed. The proposed schemes are mainly based on DEFLATE buffer latency issue. The proposed information hiding schemes embed additional data in a cover file during the compression process. So the resultant file is compressed as well as contains additional data embedded in it. The proposed schemes are explained below in detail:

Input: For both steganographic schemes two files are provided as input to the DEFLATE compression process explained below:

1. Carrier Medium: A cover file also known as carrier file to serve as the medium to hide or store secret data e.g. A text file in this case.

2. stegano_data: An encrypted secret data file to hide inside the carrier medium.

Stegano_data is most of the time encrypted to provide extra security in case intercepted by unintended recipient.

4.2.1 Scheme I

In the proposed information hiding scheme the DEFLATE compression function is forced to perform flush every time to hide additional data during compression. Z_SYNC_FLUSH mode is used here to add secret data. DEFLATE uses an input buffer for reading input from input file and an output buffer for writing output to the compressed file of same sizes. An additional buffer is used which contains the stegano_data.

The Embedding Process

The embedding process flow is mentioned below in steps:

1. Define the size of the input and output buffer and initialise DEFLATE compression parameters with an initialization function i.e. deflateinit() in this case.

2. The input buffer reads a block of data from the cover file into a fixed size buffer. The stegano_data is also read in an information hiding buffer.

3. DEFLATE calls compression function i.e. deflate () for the data present in input buffer. The flush mode provided in this case is Z_SYNC_FLUSH instead of Z_NO_FLUSH.

4. During the call of deflate () when Z_SYNC_FLUSH is performed a type 0 block(explained in detail in section 3.2.1) of stegano_data is emitted from information hiding buffer and written to the compressed output file.

5. Deflate () is repeatedly called as long as all data from the input file is read and compressed. During each call of compression function an additional block of stegano_data is embedded in the output file at the time of Z_SYNC_FLUSH.

At the end of input file the flush mode is set to Z_FINISH to successfully end the compression function.

3- bit Header	2-byte Length of block	2-byte One's Compliment of length of block	The data part

Figure 4.1: Type zero block format

Output: The output of the embedding process is a DEFLATE compressed file of Zlib format (as explained in section 3.4.1)with additional stegano_data file embedded in it.

The Extraction Process

The data embedded during compression process can be extracted before decompression of the output file with a special function. Every type 0 block embedded into the compressed file has a specific format represented in figure 4.1.

As the additionally embedded type 0 block of stegano_data is not compressed during the DEFLATE compression process. The data can be extracted by observing this specific format in the output compressed file. The extraction process flows as follows:

1. The data embedded file is opened into binary mode and is read byte by byte into a buffer completely.

2. The buffer is searched for special patterns containing a 2-byte value with next two bytes as its one's compliment. The above pattern when found is assumed to be start of the embedded data block.

3. As the length of the embedded data block is known in advance, it is extracted from the compressed file starting ahead of the one's compliment value to the length of the embedded block value and is written to a new file.

4. The extraction process searches the complete buffer to find and extract all blocks of embedded data.

The output of the extraction process consists of two files as mentioned below:

1. A clean compressed cover file which can be decompressed by using DEFLATE decompression function.

2. The stegano_data file, containing secret encrypted data.

Explanation

As it is already stated that scheme I embeds data in a file during compression when flush is performed. DEFLATE compression parameters and data structures are initialized before the start of compression. The stegano_data file is read into a information hide buffer which is embedded into cover file block by block during its compression. The compression function deflate() is provided with information hide buffer and the block length to be hidden. As the size of the output file containing stegano_data must be less than the original cover file. The optimal length value of the each block to be embedded is defined with reference to the input and output buffer size as given below:

$$\text{Hide\_len} = 20*\text{CHUNK}/100$$

Where
Hide_len : length of block to be embedded
CHUNK: size of the input/output buffer
Blocks of the above length value are embedded in cover file during each flush operation. During DEFLATE compression when flush such as Z_SYNC _FLUSH is performed.Instead of an empty type 0 block (00 00 FF FF) a stegano_data block of specific length is emitted and written to the output file.The compression process resumes after flush and continue as long as end of file not reached.After end of compression process the output file produced has uncompressed stegano_data embedded in it. Which can be extracted by running the extraction procedure on the output file. The extraction process searches for the type 0 blocks of specific length embedded in the compressed output file and extracts them.

4.2.2 Scheme II

The scheme II is based on the same principle as the scheme I in a way that secret data block is embedded at the time when flush operation is performed during file compression. Since in scheme I the embedded data is not compressed but only encrypted and is vulnerable to detection in case intercepted by unintended recipient. This weakness is covered in scheme II. In scheme II the additionally embedded data is also compressed with the cover file to make its presence invisible to the third party and it does not add any information about the length of the stegano_data block added in the cover file. Instead the length of embedded block is pre shared among the communication parties. Here for convenience the embedded block length is taken same as the size of input and output buffer.

The Embedding Process

The flow of embedding process is as follows:

1. Define the size of the input and output buffer and initialise DEFLATE compression parameters with an initialization function i.e. deflateinit () in this case.

2. The input buffer reads a block of data from the cover file into a fixed size buffer.

3. DEFLATE calls compression function i.e. deflate () for the data present in input buffer. The flush mode provided in this case is Z_SYNC_FLUSH instead of Z_NO_FLUSH.

4. Since the flush operation is performed at the end of each call of deflate (). So after Z_SYNC_FLUSH is performed the input buffer reads data block from the stegano_data file instead of cover file and compresses it.

5. In this way the data from cover file is compressed block by block and during the process after every Z_SYNC_FLUSH call the data block from stegano_data is read in input buffer and compressed. This process repeats till all the data from the cover file is compressed.

Hence all the data from the cover file and stegano_data file is compressed in alternative blocks in the output file. The output of the embedding process is

1. A DEFLATE compressed file of Zlib format (as explained in section 3.4.1)with additional stegano_data embedded in it i.e. steganographic cover file.

The Extraction Process

The stegano_data embedded by the above scheme can be extracted during the process of decompression of the cover file. The decompression function of DEFLATE i.e. INFLATE is called to decompress the cover file. Since the data is embedded in alternative blocks of cover file and stegano_data file so it is extracted in the same way. To correctly extract the embedded stegano_data the input/output buffer size or CHUNK value for extraction process must be same as embedding process. So the I/O buffer size can act as a key to secure the embedded data block length. The extraction process is discussed in detail below:

1. Define the size of the input and output buffer and initialise DEFLATE decompression parameters with an initialization function i.e. inflateinit() in this case.

2. The input buffer reads a block of data from the compressed cover file into a fixed size buffer.

3. DEFLATE calls decompression function i.e. Inflate () for the data present in input buffer. The decompressed data is written to the output buffer. Since the data from two sources was compressed together in alternative blocks. The stegano_data can be separated from the decompressed cover file data.

In this way the data from cover file is decompressed block by block. This process repeats till all the data from the cover file is decompressed. The output of the extraction process consists of two files as mentioned below:

1. A clean decompressed cover file.

2. The stegano_data file, containing secret encrypted data.

Explanation

Scheme I and II have similar basic information hiding principle but varies in few parameters. In scheme II the stegano_data is also compressed with the cover file. The data to be compressed is read from cover file into input buffer and passed to deflate () for compression block by block. During compression after flush is performed in next call the data block is read from stegano_data file, compressed and written to the output file. In this way data blocks are alternatively read from cover file and stegano_data file into input buffer and are compressed in a single file. Currently in this scheme the length value of stegano_data block and cover file data block is taken same. The resultant compressed cover file produced contains both stegano_data and cover file data. The stegano_data can be extracted by decompressing the output file. For correct decompression of steganographic cover file the size value of the input and output buffer for compression and decompression processes should be same otherwise the embedded data could not be retrieved correctly. Hence the size value of input and output buffer serves as a key to correctly extract the embedded stegano_data.

Chapter 5

Implementation and Evaluation

This chapter includes the implementation or coding details of the steganographic schemes presented in chapter 4 and the results after verification and testing of the proposed schemes.

5.1 Implementation Details

DEFLATE is a popular compression algorithm used in many archive formats and data communication protocols for data compression. Zlib is a free open source lossless compression library available on internet. It is a primary implementation of the DEFLATE compression scheme. It was developed by Jean Loup Gailly (Compression) and Mark Adler (decompression). It provides flexibility of using DEFLATE compression functions in terms of adjusting compression parameters as per requirement.
For coding the proposed information hiding schemes Zlib version 1.2.5 is taken as a source for DEFLATE implementation. The Zlib code for DEFLATE compression and decompression functions is modified by adding new functions for the purpose of information embedding and extraction. Coding is done using C source code of Zlib 1.2.5 with the compiler gcc in ubuntu Environment.

5.2 Evaluation and Testing Results of the Proposed Schemes

Since the proposed information hiding schemes are based on flush operations. It is necessary to know that how much flushes are performed during compression and how number of flush affects the size of embedded data. In following pages the information embedding capacity of the proposed scheme is explained and an optimal threshold value for the embeddable data size with respect to compressed steganographic cover file size is defined.

5.2.1 Number of Flushes

The amount of Data embedded using Scheme I and II entirely depends on the number of flushes performed during file compression. For very large input and output buffers the compression process is performed in one step and buffer latency is not an issue therefore, the flush mode used here is Z_NO_FLUSH i.e. no flush. Since data hiding schemes embed data only when flush is performed so it is preferable to use small size input and output buffers. It is noted that number of flushes increase with small buffer size. In this way compression would be done in many steps of reading input data into input buffer and writing it to output buffer after compression and thus increasing the amount of data embedded. It is not possible to count the exact number of flushes performed during compression process but the amount of embedded data can be controlled by using other parameters. A graph is presented in figure 5.1 showing a direct relationship between amount of hidden data and number of flushes. According to that with the increase of number of flushes the amount of data that can be embedded also increases. Since it is observed through experiments with multiple files that forcing a flush during compression process does not degrade compression ratio badly so compression parameter adjustment can be avoided during information hiding.

5.2.2 How much data can be embedded?

A good information hiding scheme can embed data as long as the size of compressed data file having hidden data is less than the original uncompressed cover file so that a user can not suspect the hidden data inside. Scheme I and Scheme II can embed large amount of data keeping the size of compressed cover file less than the original size. Figure 5.2 presents a graph showing that the size of steganographic cover file increases with increase in number of flushes during compression. The data embedding capacity for scheme II is much greater than Scheme I. Since in both methods data is embedded in blocks, after verifying the schemes the optimal value for the size of each block embedded using scheme I is mentioned below:

Hide_len = 20*CHUNK/100

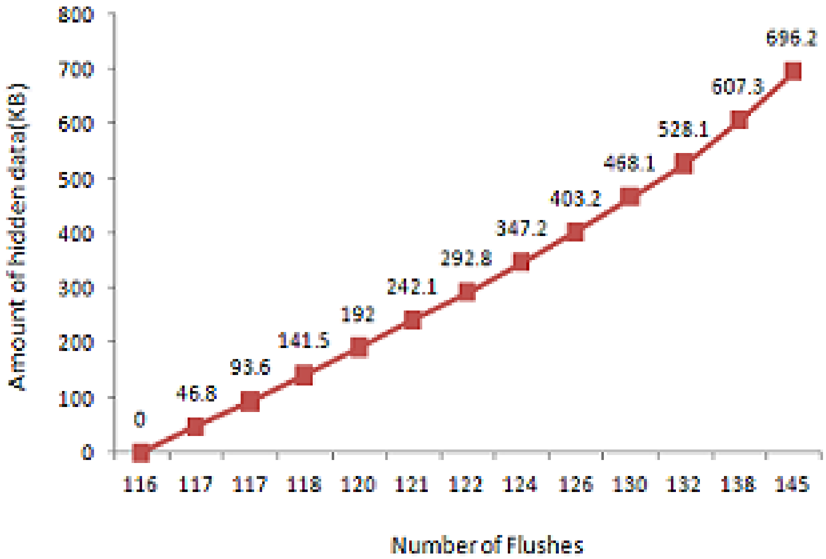

Figure 5.1: A direct relationship in amount of hidden data with no of flushes,info hidden using z_sync_flush

Where
Hide_len: Length of block to be embedded
CHUNK: Size of the input/output buffer
The value of Hide_len largely depends on the size of input/output buffers namely CHUNK value. For small buffer size value the Hide_len value up to 45% of CHUNK size can keep the data hidden compressed cover file size small than the original uncompressed cover file. On the other hand for large buffers the safe Hide_len value can exceed to 55% of the CHUNK value still keeping the data hidden compressed cover file size small than the original uncompressed cover file as presented in table 5.1 in section 5.3.1. Even larger values of the Hide_len can increase the size of the compressed cover file than the original uncompressed cover file.

5.3 Verification of the Proposed Schemes

The proposed schemes have been verified by hiding data in several text and doc files during their compression. The proposed schemes work well for both type of files. The detail is described below:

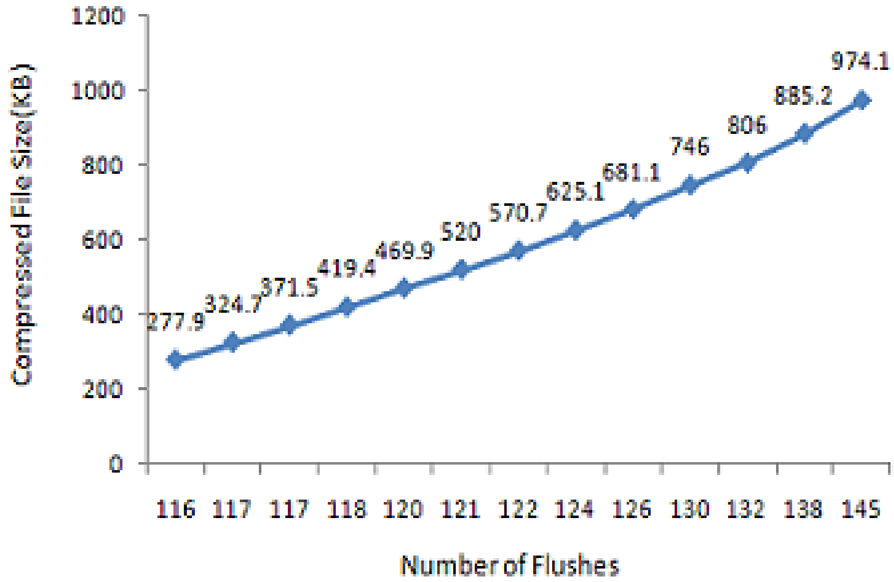

Figure 5.2: A direct relationship in no of flushes with compressed file size,info hidden using z_sync_flush

5.3.1 Scheme I

Scheme I hides a block of data in a file every time when flush is performed during compression. The flush mode can be Z_SYNC_FLUSH and Z_PARTIAL _FLUSH. Scheme I was used to embed data in different doc and text files. Scheme I is verified by embedding different amount of information in a cover file in table 5.1. The original cover file size is taken to be 934.5KB and the input/output buffer size to be used during compression is taken to be 8192bytes. Table 5.1 shows that how amount of hidden data increases with number of flushes and shows that for a buffer of size 8192bytes the optimal size for the data block to be embedded is 55% of the CHUNK. For larger values of embedded data block the size of the steganographic cover file increases from original uncompressed cover file. The table 5.1 shows that by increasing number of flushes more data can be hidden inside a compressed file thus increasing the size of compressed file. The size of data to be hidden should not increase the steganographic cover file size than the uncompressed cover file size. Since an ordinary user is not concerned about the compression ratio or unaware about it and considers a compressed file correct as long as compressed file size is less than the original uncompressed cover file size. It is evident from scheme I statistics table that for a 934.5KB cover file the

Amount of hidden data			-
Embedded Block size	Hidden Data Size (Kb)	Size of Compressed File (KB)	No of flushes
0	0	277.9	116
5% of CHUNK	46.8	324.7	117
10% of CHUNK	93.6	371.5	117
15% of CHUNK	141.5	429.4	118
20% of CHUNK	192	469.4	120
25% of CHUNK	242.1	520	121
30% of CHUNK	292.8	570.7	122
35% of CHUNK	347.2	625.1	124
40% of CHUNK	403.2	681.1	126
45% of CHUNK	468.1	746	130
50% of CHUNK	528.1	806	132
55% of CHUNK	607.3	885.2	138
60% of CHUNK	696.2	974.1	145

Table 5.1: Scheme I statistics, Cover file size: 934.5 KB, CHUNK: 8192B

Cover file Size	Compressed file size	steganographic cover file size
2.5 MB	17.6 KB	35.1 KB
3.9 MB	27.4 KB	55.9 KB
4.6 MB	32.2 KB	65.0 KB
7.5 MB	52.1 KB	105.7 KB
8.0 MB	55.6 KB	112.8 KB
8.5 MB	59.3 KB	120.3 KB

Table 5.2: Scheme II statistics, CHUNK (size of I/O buffers): 16384Bytes

amount of data that can be hidden safely is 607.3KB. So the data hiding capacity for scheme I is more than half of the cover file size. Files with embedded data using scheme I are studied with hexeditor to have a finer look on how a compressed file with additional data embedded looks like in figure 5.3 and figure 5.4 in section 5.5.

5.3.2 Scheme II

The information hiding capacity for scheme II is greater than scheme I. The information hiding capacity has a direct relation with number of flushes because data is hidden or embedded when a flush is performed. Since data from cover file and stegano_data_file is mixed and then compressed it increases the information embedding capacity to much greater extent e.g. A cover file of size 68.5KB can hide another 68.5KB file inside it during compression with alternative flushes still keeping the compressed information hidden output file's size less than the original uncompressed cover file size. E.g. A cover file of size 68.5KB can be compressed to a size of 7.77KB with additional compressed 68.5KB hidden in it. Table 5.2 presents information embedding statistics for scheme II using Z_SYNC_FLUSH: Hence using scheme II data from multiple files can be mixed and embedded inside a cover file and since the embedded data is also compressed so it is not visibly differentiated by any unintended recipient if the compressed file is studied with any file editor i.e. hex editor as shown in figure 5.5 in section 5.5.

5.4 Scheme I Vs Scheme II

Table 5.3 presents a comparison between scheme I and scheme II in terms of security and information embedding capacity. It shows scheme II is more

SCHEME I	SCHEME II
Hidden data not compressed.	Hidden data is compressed with cover file data.
vulnerable to confidentiality loss.	steganographic cover file Not vulnerable to confidentiality loss
information embedding capacity is less than the original cover file size.	information embedding capacity equal to the size of original cover file.
The cover file can be retrieved half or less but not full by decompression.	Cover file is not damaged and can be retrieved full by decompression.

Table 5.3: Scheme I Vs Scheme II

secure than scheme I and provides much greater information embedding capacity.

5.5 Steganographic Cover Files studied with Hex editor

Hex dumps of different steganographic cover files compressed using scheme I and scheme II are presented in this section. Figure 5.3 and 5.4 shows hex dumps of the steganographic cover files for scheme I. Since here embedded data block is not encrypted for visibility. Figure 5.5 contains hex dump of the steganographic cover file compressed using scheme II.
In figure 5.3 a steganographic cover file compressed using scheme I is given. Since it is already discussed that in scheme I the embedded data block is not compressed with the cover file data. So the hex dump of the file clearly shows the start of embedded data block inside the cover file.In figure 5.3 stegano_data is embedded using Z_SYNC_FLUSH. The starting two bytes of the highlighted embedded data block in the figure tells the length of the data block and next two bytes are one's compliment of the length value. After these four bytes the actual stegano_data block is stored.In this way multiple stegano_data blocks are hidden inside the steganographic cover file.
Figure 5.4 shows a text steganographic cover file with data embedded using Z_PARTIAL_FLUSH using scheme I.The length of embedded data block is large so it is not covered in the figure completely.For Z_PARTIAL_FLUSH length of the data block is not stored inside the file and only data part is stored.

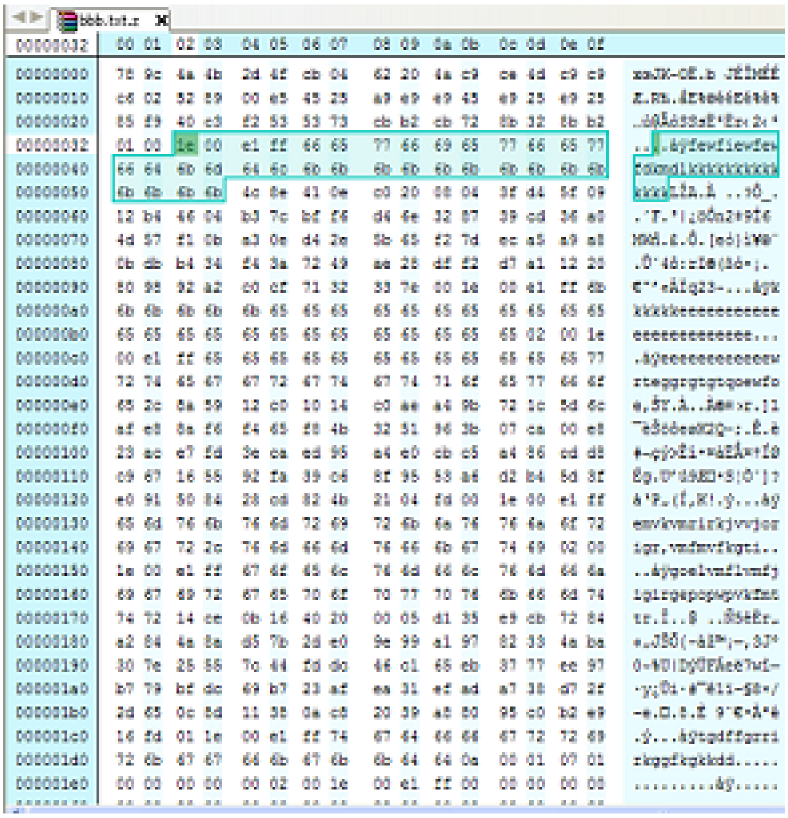

Figure 5.3: Steganographic cover file with info block hidden using z_sync_flush using scheme I, arrow pointing start of hidden data

Figure 5.5 shows hex dump of a steganographic cover file compressed using scheme II. The original size of the file is 7.5MB which can be compressed to 52.1 KB and with hiding another file of 7.5MB inside it the resultant steganographic cover file is produced of size 105KB. Since scheme II compresses stegano_data block with the cover file data so hidden data can not be visibly differentiated in the steganographic cover file even if studied with any hexeditor. It is said that steganography is basically security through obscurity so it shows clearly that hidden data can not be differentiated from the cover file data.

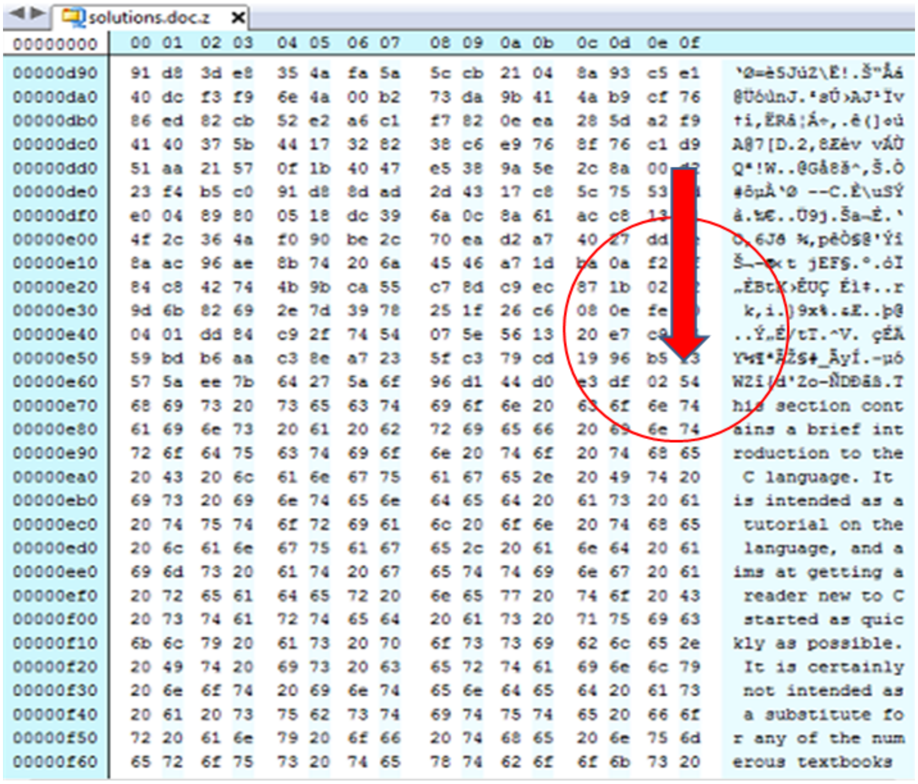

Figure 5.4: Steganographic cover file with info block hidden using z_partial_flush using scheme I, arrow pointing start of hidden data

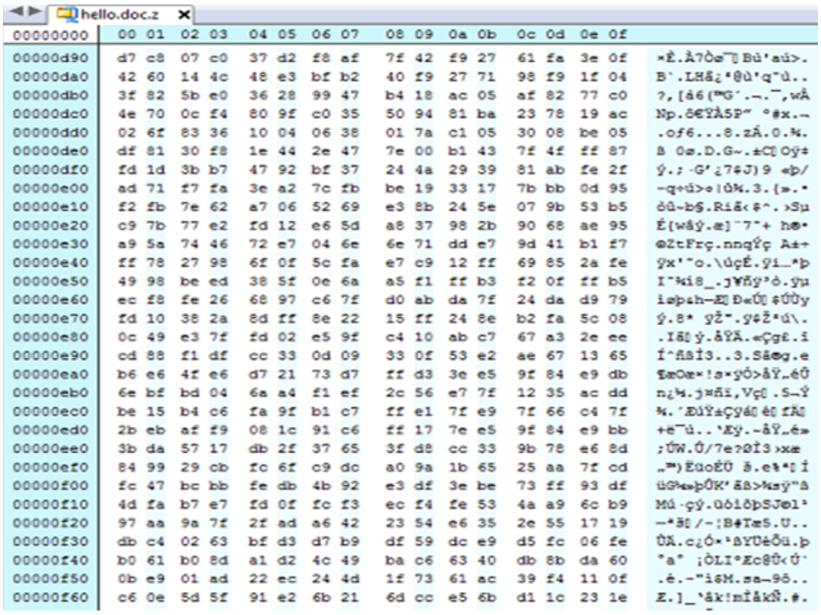

Figure 5.5: Steganographic cover file with info block hidden using z_sync_flush using scheme II, hidden data cannot be differentiated

Chapter 6

Conclusion and Future Work

This chapter concludes the whole document and explains the future research intentions of the researcher.

6.1 Conclusion

Steganography is an ancient art used to hide information in different communication media. Steganographic model includes a sender, carrier medium to hide data, and a receiver. The most commonly used media for information hiding include text and doc files, images, sound,and zipped archives. Exploring steganography options in compressed archives is not very new. Algorithms have been developed to hide additional data in zipped archives either by exploiting file format architecture or modifying the compression algorithm to embed additional data inside a file during compression but these schemes limit in information embedding capacity and are vulnerable to detection. Thus, new information hiding schemes are proposed which embed data by exploiting the compression algorithm implementation. DEFLATE is a widely used compression algorithm in compression tools and various communication protocols. Its Zlib implementation is studied for the purpose of information hiding options. On the basis of discovered vulnerabilities in DEFLATE two schemes of information hiding are proposed. The basic principal is to embed additional secret information inside a cover file during compression when a flush operation is performed. DEFLATE uses different flush operations to avoid buffer latency during compression. These flush modes write some extra data to the compressed output file. User can hide additional data on these places where bytes are added by flush operations during compression. The proposed schemes are evaluated for hiding information in text and doc files and are proved to provide very good information hiding capacity and do not compromises the compression speed. Scheme II is proved to be better than scheme I.

6.2 Future Work

DEFLATE is a widely used compression algorithm in most of the communication protocols for data compression. Most common protocols that use DEFLATE compression are PPP, TLS, SSH, IP, and HTTP. In these protocols IP compression does not use flushing but in others compression is done with flushing e.g. In Point to Point protocol (PPP) data is communicated in the form of packets. A packet can contain one or many DEFLATE blocks and a packet ends at Z_SYNC_FLUSH (or Z_FULL_FLUSH) and at the end of block a sequence of 00 00 FF FF is emitted. In this way TLS, SSH and HTTP use some kind of flushing too. Presently, the proposed schemes work to hide information in a single file using flush modes during compression but the information hiding process can be extended to embed and secretly communicate information using internet communication protocols between two parties on web. Above mentioned protocols are documented using DEFLATE for compression. The future research intentions include:

- Analysing PPP, TLS, SSH, HTTP for secretly information embedding and transmission in interactive web applications.
- Defining a model of secret information sharing between two parties using DEFLATE compression in transmission protocols.

Bibliography

[1] Neil F.Johnson, Sushil Jajudia ,“ Exploring steganography, Seeing the unseen”, George Mason University,IEEE Computer, Feb. 1998.

[2] A. Kumar, K. M. Pooja, “steganography:A Data Hiding Technique”, International journal of Computer Applications, Vol. 9, 2010.

[3] M. Voksan, T. pericin, B. Carney, “Hiding in the familiar: steganography and vulnerabilities in popular archive formats”, Black Hat Europe 2010, Barcelona.

[4] Steganography 16-Hiding additional files in a ZIP archive. http://www.codeproject.com/KB/security/steganodotnet16.aspx. (last visited in November 2011).

[5] K N Chen, C.F. Lee, C-C Chang, H-C Lin, “Embedding Secret message using modified Huffman Tree”, International Conf. on Intelligent Information Hiding and Multimedia Signal Processing,2009.

[6] Huffman Coding Example. http://www.binaryessence.com/dct/en000080.htm. (last visited in December 2011)

[7] An explanation of the DEFLATE Algorithm. http://zlib.net/feldspar.html. (Last visited in December 2011)

[8] Deflate compressed data format specification version 1.3. http://www.ipgz.org/zlib/rfc-deflate.html. (last visited in November 2011).

[9] Zlib compressed data format specification version 3.3. http://www.gzip.org/zlib/rfc-zlib.html. (Last visited in January 2012).

[10] Zlib flush modes. http://www.bolet.org/~pornin/deflate-flush-en.html. (Last visited February 2012).

[11] GZIP file format specification version 4.3. http://www.gzip.org/zlib/rfc-gzip.html. (Last visited january 2011).

[12] K yoshioka, K Sonoda, O Takizawa, "Information Hiding on lossless data compression",International Conf. on Intelligent Information Hiding and MultimediaSignal Processing, 2006.

[13] K N Chen, C.F. Lee, C-C Chang, H-C Lin, "Embedding Secret message using modified Huffman Tree", International Conf. on Intelligent Information Hiding and Multimedia Signal Processing 2009.

[14] Zip/JPEG mask and encryption. http://www.sfu.ca/~vwchu/zjmask.html. (Last visited in january 2011).

[15] Zlib 1.2.5 Manual. http://zlib.net/manual.html. (Last visited November 2011).

[16] Hypertext Transfer Protocol-HTTP/ 1.1. http://tools.ietf.org/html/rfc2616. (Last visited in January 2012).

[17] PPP Deflate Protocol. http://www.ietf.org/rfc/rfc1979.txt. (Last visited in January 2012).

[18] Transport Layer Security Protocol Compression Methods. http://www.ietf.org/rfc/rfc3749.txt. (Last visited in January 2012).

[19] The Secure Shell (SSH) Transport Layer Protocol. http://www.ietf.org/rfc/rfc4253.txt. (Last visited in January 2012).

[20] IP Payload Compression Protocol (IPComp). http://www.ietf.org/rfc/rfc2393.txt. (Last visited in January 2012).

[21] IP Payload Compression Using DEFLATE. http://www.ietf.org/rfc/rfc2394.txt. (Last visited in January 2012).