

LINGUISTIC STEGANOGRAPHY:
PASSING COVERT DATA USING
TEXT-BASED MIMICRY

by

Adam Tenenbaum

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE

Supervisor: D. Kundur
April 2002

Abstract

The goal of linguistic steganography systems is to transmit a secret message over an open communication channel while concealing the presence of the secret message altogether. The secret message is hidden by encoding its bits within a “cover” message that mimics natural language. Existing text mimicry algorithms are flawed in that there exists a tradeoff between the quality of the output text and the resources required to manually design an appropriate grammar for the content of the cover message.

In Peter Wayner’s basic mimicry algorithm [11, Chapter 6], the system learns from frequency analysis of a “training source” in order to attempt to mimic the source. This thesis improves upon Wayner’s algorithm by changing the “atom” in frequency analysis from a single character to a single word. The resulting linguistic steganography algorithm generates a cover text that more closely resembles the style of the training source but also mimics the grammar of the source text in a dynamic, automated fashion.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Deepa Kundur. She was always willing to give me a nudge in the right direction while still letting me discover my own path.

I would also like to thank Peter Wayner for providing an early manuscript of the second edition of his book, *Disappearing Cryptography*. This book proved to be an invaluable resource and also inspired this project.

I would like to thank my parents, Linda and Jerry Tenenbaum, for their emotional and financial support for the past four years of my education. I also thank the rest of my family and friends for being interested (or at least pretending) when I rambled on and on about steganography. I particularly owe thanks to my roommate, Roni Gordon, for being a good sounding board when I was first developing some ideas presented in this thesis and for **actually** being interested.

Finally, and most importantly, I would like to thank Laura Charbonneau for her unwavering love and support throughout this past term. Writing a thesis is a stressful endeavour, and it takes someone special to put up with an often frustrated author like myself.

Table of Contents

1	Introduction	1
1.1	Research Motivation	1
1.1.1	Encryption	2
1.1.2	Data Hiding: A Viable Alternative	2
1.1.3	Mimicry: A Different Kind of Steganography	3
1.2	Research Objectives	4
1.2.1	Dynamically Generated Grammars	4
1.2.2	Hiding the Secret Message	4
1.2.3	Designing a Practical Algorithm	5
2	Background	6
2.1	A Brief History of Steganography	6
2.2	Outline of a Steganographic System	8
2.3	Steganography vs. Cryptography	8
2.4	The Strength of Steganography	9
2.4.1	Frequency Analysis	9
2.4.2	Subjective Analysis	9
2.5	Why Use Mimicry?	10
2.5.1	The One-time Pad	11
2.5.2	Existing Mimicry Algorithms	11
2.6	Mimicry Algorithms that Use Static Grammars	12
2.6.1	Context Free Grammars	12
2.6.2	NICETEXT	13
2.7	mimic: A Dynamic Mimicry Algorithm	15
2.7.1	Huffman Codes	17
2.7.2	Optimal Codes	18

2.7.3	Huffman Coding Algorithm	19
2.8	Using Reverse Huffman Codes to Hide Data	20
3	A New Algorithm for Mimicry-Based Linguistic Steganography	22
3.1	Formulating a New Mimicry Algorithm	22
3.1.1	Using Words as the Atomic Unit for Mimicry	22
3.1.2	Preserving the Case of Letters within Each Word	23
3.1.3	Special Considerations for Punctuation Characters	24
3.2	Mimicry Algorithm Implementation Details	25
3.2.1	Encoding: Overview	25
3.2.2	Hash Tables and Huffman Trees: Building the Frequency Model of the Source Text	26
3.2.3	Converting Characters to Bits	28
3.2.4	Producing the Stegotext	29
3.2.5	Decoding: Overview	31
3.2.6	Using the Frequency Model to Reconstruct the Stream of Bits from the Stegotext	31
3.2.7	Converting Bits to Characters	32
4	Analysis	33
4.1	Subjective Analysis	33
4.1.1	Selection of Test Cases	33
4.1.2	Results	34
4.2	Frequency Analysis	34
4.2.1	Selection of Test Cases	35
4.2.2	Results	35
4.3	Testing Various Parameters of the Mimicry Engine	40
4.3.1	Expansion Factor vs. Order	40
4.3.2	Expansion Factor vs. Message Length	43
4.3.3	Timing Tests	46
5	Conclusions and Future Research	49
5.1	Conclusions	49
5.2	Future Research	50

Bibliography	52
A References	54
B Readability Survey	57
B.1 Introduction	57
B.2 Survey Contents	57
C Program Documentation	59
C.1 mimic	59
C.2 freqdist	60
D Source Code	61
D.1 mimic.c	61
D.2 mimic.h	77
D.3 hash.c	79
D.4 hash.h	85
D.5 huffman.c	87
D.6 huffman.h	91
D.7 error.c	93
D.8 error.h	93
D.9 bool.h	94
D.10 freqdist.c	94

List of Symbols

V	Secret message in steganography system, p. 8
U	Cover message in steganography system, p. 8
K	Key in steganography system, p. 8
V'	Recovered message in steganography system, p. 8
S	Source text to train upon in mimicry algorithm, p. 15
T	Output text of mimicry algorithm, or stegotext, p. 15
n	Order of the mimicry, p. 16
\bar{L}	Average codeword length, p. 17
l_i	Length of codeword i , p. 17
$H(x)$	Source entropy, p. 18
S_x	Source symbol from set S , p. 18
$p(x)$	Probability of symbol S_x occurring, p. 18
B	Stream of bits, p. 20
b_i	Single bit in a stream of bits, p. 20
M	Secret message being hidden in mimicry algorithm, p. 25
W_i	Word used in encoding or decoding of stegotext, p. 27
$n_o(i)$	Number of occurrences of a word in the frequency table, p. 27
E	Expansion factor, p. 40

List of Figures

2.1	A drawing of the San Antonio River that conceals a secret message	7
2.2	Model for steganography system	8
2.3	Frequency distribution of English language	10
2.4	Example of a context free grammar	12
2.5	Peter Wayner's mimic algorithm	16
2.6	Kraft inequality equation	17
2.7	Definition of Entropy	18
2.8	Example of Huffman coding tree	20
3.1	Modified mimicry algorithm	23
3.2	Example of Huffman coding tree using words as the atomic unit	29
4.1	Frequency distribution of stegotexts generated using <code>Lab3.txt</code> as the source text	36
4.2	Frequency distribution of stegotexts generated using <code>chapter5.txt</code> as the source text	37
4.3	Frequency distribution of stegotexts generated using <code>macbeth.txt</code> as the source text	38
4.4	Frequency distribution of stegotexts generated using <code>inferno.txt</code> as the source text	39

4.5	Testing the effect of changing order on expansion factor E	42
4.6	Testing the effect of changing secret message length on expansion factor E	45
4.7	Mimicry timing results for various source texts	47
A.1	Sample source text used to generate Huffman coding tree in Figure 3.2	54

List of Tables

2.1	Frequency distribution of symbols used in Figure 2.8 . . .	19
2.2	Reverse Huffman Codes example	21
3.1	Punctuation characters that receive special treatment in the mimicry algorithm	24
4.1	Files used as source texts in analysis of the mimicry algorithm	35
4.2	Secret message files and their corresponding sizes, used to test the effect of changing secret message length on the ex- pansion factor E	44
A.1	Frequency distribution of English language	55
A.2	Average timing test results, Lab3.txt	55
A.3	Average timing test results, chapter5.txt	56
A.4	Average timing test results, macbeth.txt	56
A.5	Average timing test results, inferno.txt	56
C.1	Command line parameters for <code>mimic</code>	59

Chapter 1

Introduction

“In the past, if the Government wanted to violate the privacy of ordinary citizens, it had to expend a certain amount of effort to intercept and steam open and read paper mail, and listen to and possibly transcribe spoken telephone conversation. This is analogous to catching fish with a hook and a line, one fish at a time. Fortunately for freedom and democracy, this kind of labor-intensive monitoring is not practical on a large scale. Today, electronic mail is gradually replacing conventional paper mail, and is soon to be the norm for everyone, not the novelty it is today. Unlike paper mail, E-mail messages are just too easy to intercept and scan for interesting keywords. This can be done easily, routinely, automatically, and undetectably on a grand scale. This is analogous to driftnet fishing — making a quantitative and qualitative Orwellian difference to the health of democracy.” [12]

**Philip Zimmerman, author of PGP (Pretty Good Protection)
June 26, 1996**

1.1 Research Motivation

As the focus of power in the world has shifted from material goods to information, the importance of maintaining the privacy of information has become crucial. As such, governments and industries have developed strong encryption technologies in order to protect their proprietary information. Unfortunately, as Philip Zimmerman has described, the “ordinary citizen’s” right to privacy has been neglected in the transition to electronic communications. Several solutions to this problem have been developed, including Zimmerman’s own **PGP** (public key encryption software). However, legal issues have arisen as a result of the development of publically available encryption technologies, since “gov-

ernment agencies here and around the world often view strong encryption technology as a military weapon.” [7]

As governments around the world lean towards this position, two options are left for those who still wish their electronic communications to be secret: continue using encryption tools such as PGP, or find an alternate method for keeping information private.

1.1.1 Encryption

The first option, continuing to use a system such as public key encryption, still poses a very real possibility for many in today’s world. The tools are freely available (although often illegal to export outside of the United States) and quite powerful. The problem with using encryption tools is that protocols such as PGP can easily be identified as encrypted — even by a machine. The encrypted messages often do not follow a normal statistical distribution (of characters in the text), and also often contain a protocol-specific signature. As a result, even “ordinary” users who simply wish to ensure the privacy of their communications may be flagged as potential trouble-makers since they are trying to conceal the contents of their communications.

1.1.2 Data Hiding: A Viable Alternative

The most likely candidate for an alternative method of sending secret messages across open communication channels is known as data hiding, or *steganography*. Steganographic methods do not focus on concealing the **meaning** of the secret message. Rather, in steganography, the very **presence** of the secret message is concealed within other data — the *cover message*.

Almost every medium has been used as a cover message for steganography: images, sound samples, markup languages (such as HTML and XML) and even plain text. The last of these, plain text, is of particular interest as it is the medium used for the dominant form of communication on the Internet — e-mail. This process of hiding secret information within plain text is also known as *linguistic steganography*.

Regardless of the medium being used, traditional steganographic systems have one common goal: to hide covert information without causing perceivable changes in the cover

message. Several existing methods take advantage of weaknesses in the human sensory systems by modifying the least significant bits in image or sound files in order to encode hidden information. The changes are subtle enough that they cannot be detected by the human senses; however, a recipient who knows of the presence of the secret message (and also knows the algorithm and key used to do the hiding) can easily decode the secret information with the aid of a computer program. Similar systems also exist for plain text which manipulate the *whitespace* (spaces, tabs, and carriage return / line feed characters) to encode information. Unfortunately, all of these systems suffer from the same weakness: if multiple messages that use the same cover message are intercepted, the strength of the steganography may be weakened.

1.1.3 Mimicry: A Different Kind of Steganography

While traditional steganographic methods hide information by altering existing cover messages, an alternative method known as *mimicry* hides data in a different manner. In mimicry-style steganography, a program generates the stego object (an encoding of secret data within cover data) so that it will appear to be something other than the secret message. For example, mimicry-based graphical steganography may generate fractal images, hiding secret data in the least significant bits of the resulting image. A music-based system could generate random MIDI scores, where the choice, duration, and volume of notes encode data. In mimicry-based linguistic steganography, a set of grammatical rules is used to generate realistic looking text (called the *stegotext*), and the choice of each word determines how secret message bits are encoded.

Static Grammars

Traditional linguistic mimicry-based steganography algorithms use static grammars. The system's user must design a grammar that he wishes the text to mimic¹. This design process can often be tedious, and the quality of the resulting stegotext directly depends on the quality of the grammar.

Issues also arise regarding the security of the grammar. In mimicry-based linguistic steganography systems, the grammar acts as the key for hiding data. This becomes a problem when both the sender and receiver must have a copy of the grammar (to encode

¹Examples of such systems can be seen in Chapter 2.

and decode the secret message). If the secrecy of the grammar is compromised in transit from the sender to the receiver, it will need to be replaced by a different grammar, requiring another tedious design process.

1.2 Research Objectives

The goal of this thesis is to formulate, implement, and analyze the effectiveness of a mimicry-based linguistic steganography algorithm which:

1. Dynamically models a grammar using frequency analysis of a particular source text to establish a set of rules for a mimicry algorithm as well as a key for data hiding.
2. Uses the modelled grammar's rules to hide a secret message within a piece of stegotext that both:
 - Mimics the statistical distribution of the source text.
 - Mimics the grammar of the source text.
3. Is practical for everyday use. The algorithm must be both reversible and efficient.

1.2.1 Dynamically Generated Grammars

In contrast to static grammars, which require a laborious design process, this thesis proposes a method of modelling grammars in a dynamic, automated fashion using frequency analysis of a source text. A frequency model of the source text is built to establish the frequency of patterns of groups of words. This frequency model is used to establish “loose” grammatical rules. These rules do not follow the grammar of any specific language; rather, the output of the grammar-generation algorithm produces a set of rules that can be used to imitate any source text (regardless of language). The dynamic nature of this algorithm obviates the need for a lengthy design process, as a new grammar can be generated simply by changing the source text.

1.2.2 Hiding the Secret Message

The grammar that has been generated is then used to construct a mimicked message. In doing so, the mimicry algorithm hides bits of data using a data structure known as a

Huffman coding tree. The Huffman coding tree arranges its “leaves” (here, words from the source) according to the frequency of their occurrence, making it an ideal representation for a frequency-based model (See 2.7.1 for further details).

The secret message is converted into a sequence of bits (0’s and 1’s) and these bits are used to select a path when traversing the Huffman tree. The resulting output stream is a text that not only mimics the source in its grammar, but also has a similar frequency distribution.

1.2.3 Designing a Practical Algorithm

It is important for the resulting algorithm to be both reversible and efficient. Reversibility is critical, since a mimicry algorithm for which the secret message could not be extracted from the stegotext by the intended recipient would have no practical use. The algorithm must also be efficient, as an overly complicated algorithm’s usage would be limited only to those with large amounts of computing power.

Chapter 2

Background

2.1 A Brief History of Steganography

Steganography, more commonly referred to as “information hiding”, comes from the Greek roots *steganos*, meaning “covered”, and *graphein*, meaning “to write” [9]. Some of the earliest known examples of steganography were documented by the Greek chronicler Herodotus. In *The Histories*, Herodotus describes how Demaratus (fifth century B.C.) passed secret military plans of the Persian invader Xerxes to his Greek homeland without having his communication intercepted by the invading Persian guards. Demaratus describes that the message could be passed by “scraping the wax off a pair of wooden folding tablets, writing on the wood underneath what Xerxes intended to do, and then covering the message over with wax again.” [9] Other steganographic methods seen throughout history [6, 9] include:

- Encapsulating a message in a tiny wax ball which was then swallowed.
- Writing in “invisible” inks that would only appear when heated (organic substances, such as urine, milk, and vinegar) or treated with the appropriate chemical (sympathetic chemicals, such as iron sulfate and potassium cyanate).

A more recent example of steganography can be seen in the “San Antonio River drawing” (Figure 2.1), generated by the San Antonio postal censorship station. The short and long blades of grass along the riverbank are used to spell out dots and dashes in Morse Code [6].

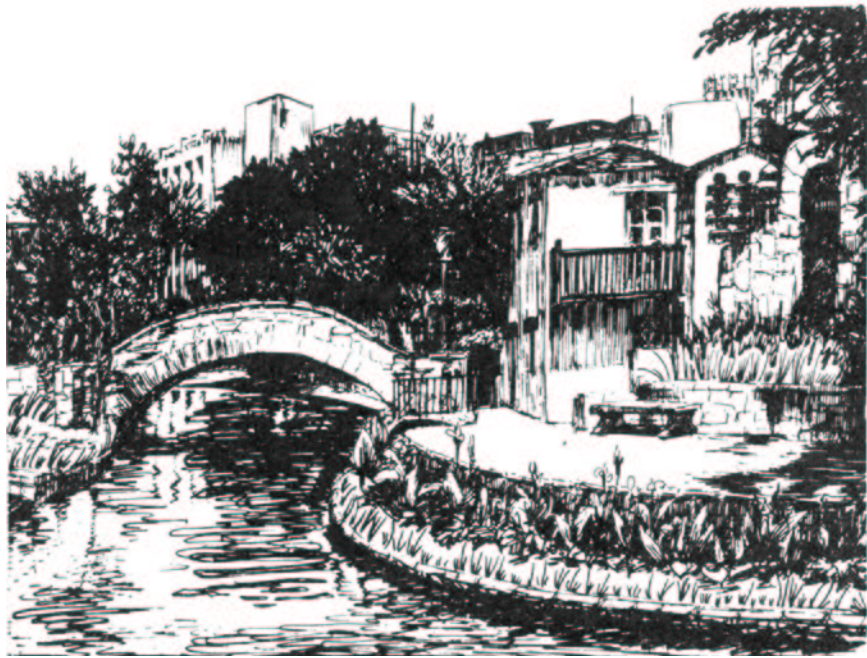


Figure 2.1: A drawing of the San Antonio River that conceals a secret message [6]

2.2 Outline of a Steganographic System

A steganographic system can be modelled as follows:

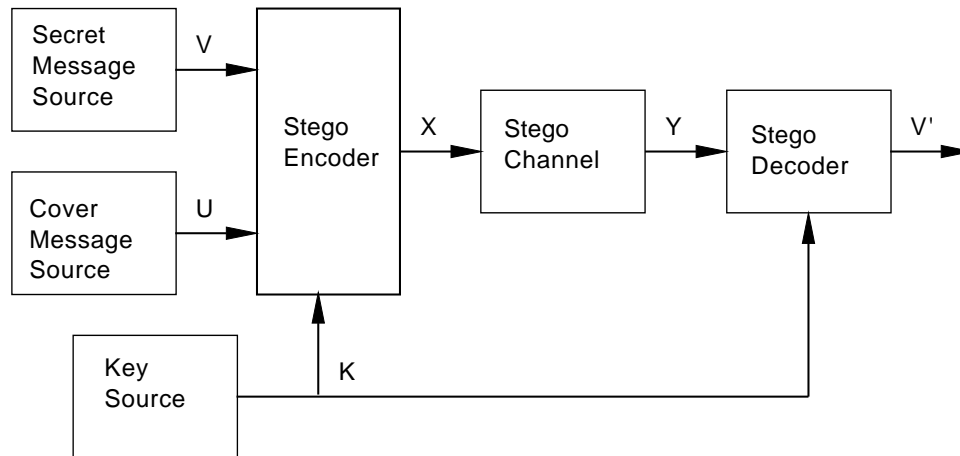


Figure 2.2: Model for steganography system [8]

Here, a secret message V is combined with a cover message U using a key K in order to create the stegotext. This stegotext is transmitted over an open stego channel where it is vulnerable to interception. Once the stegotext is received, it is decoded in combination with the key K , generating the recovered message V' .

2.3 Steganography vs. Cryptography

Steganography, by definition, is quite different from cryptography (from the Greek root *kryptos*, meaning “hidden”). Whereas cryptographic methods are used to hide the **meaning** of a message, steganographic methods hide the **existence** of the secret message itself. As such, many users are selecting steganography over cryptography, so as to not draw attention to the fact that they are sending concealed information.

Steganography and cryptography may differ in theory, but in practice, the difference between the two methods of concealing information is not always as clear-cut. Peter Wayner discusses this point in his introduction to *Disappearing Cryptography*:

Drawing a line between the two is both arbitrary and dangerously confusing. Most good cryptographic tools also produce data that looks almost perfectly random. You might say that they are trying to hide the information by disguising it as random noise. On the other hand, many steganographic algorithms are not trivial to break even after learning that there is hidden data to find. [11]

The quote is particularly appropriate in describing the algorithm developed in this thesis. The extension of Wayner's `mimic` algorithm (Section 2.7) does indeed hide a message; however, even after one has discovered that a message is hidden, it should still prove quite difficult to find the secret message without knowledge of the key used to generate the grammar. The question remains: How strong is a steganographic system?

2.4 The Strength of Steganography

Steganography is a relatively young science in academic circles. When compared to other research interests in data communications, it is evident that the development of mathematical theories for steganography lag far behind the theories relating to information theory and cryptography. At this time, no strong mathematical models for evaluating the strength of steganographic systems exist. As such, it is difficult to gauge the strength of any given steganographic algorithm. The two tools that are used within this thesis to evaluate the strength of the algorithm are frequency analysis and subjective analysis.

2.4.1 Frequency Analysis

One of the oldest means for discovering encrypted messages is frequency analysis. A piece of text can be analyzed automatically by a computer program which can determine if the text matches the frequency distribution of the standard English language (Figure 2.3). The most basic requirement that an effective mimicry algorithm must satisfy is matching the frequency distribution of the source. If this criterion is met, it is likely that the stegotext will evade detection by simple interception tools.

2.4.2 Subjective Analysis

A brief amount of time in developing this thesis was dedicated to the subjective analysis of the mimicry algorithm. While sophisticated computer tools are capable of examining

a piece of text for irregularities, the quality of this tool cannot be determined by this measure alone. The ultimate success of this algorithm would be in demonstrating that it is capable of producing text that can fool a human reader while hiding data at the same time. Methods for subjective analysis and the results are described in Section 4.1.

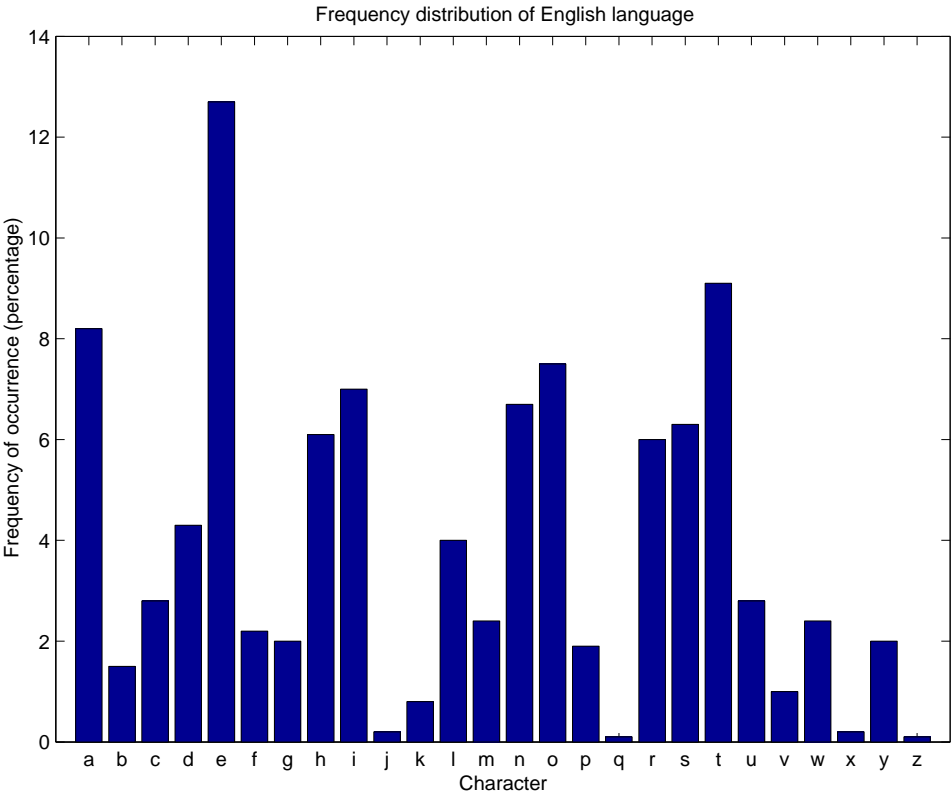


Figure 2.3: Frequency distribution of English language (see Table A.1)

2.5 Why Use Mimicry?

Many existing steganographic systems use an existing cover message and hide data by modifying information about the cover message in an “imperceptible” manner. However, these systems often suffer from a lack of robustness. For example, in a system that hides information within plain text by modifying the amount of whitespace, the hidden information may be lost if the text is reformatted (a common operation when transferring files from Windows to UNIX environments). Since such methods hide information within imperceptible changes, a user may unwittingly (or intentionally) destroy the hidden information without altering the appearance of the cover message. In contrast, a

mimicry-based steganography system encodes the hidden data as the cover message itself. As such, changes in whitespace or formatting do not affect the receiver's ability to decode the secret message.

More importantly, the use of mimicry for steganographic purposes provides additional strength when hiding information since a cryptographical element is added to the encoding process. In many traditional steganography systems, the secret message is simply hidden in perceptually insignificant bits in the cover message. Such systems that rely only on the secrecy of the algorithm for their security (i.e. ones where the algorithm is the only key) often fail once the algorithm is revealed. In contrast, the algorithm proposed within this thesis includes a cryptographical element, as the source text (that is being mimicked) is used to approximate one of the strongest forms of cryptography: the one-time pad.

2.5.1 The One-time Pad

Simon Singh describes the one-time pad as follows:

The security of the one-time pad cipher is wholly due to the randomness of the key. The key injects randomness into the ciphertext, and if the ciphertext is random then it has no patterns, no structure, nothing the cryptanalyst can latch onto. In fact, it can be mathematically proved that it is impossible for a cryptanalyst to crack a message encrypted with a one-time pad cipher. In other words, the one-time pad is not merely believed to be unbreakable, just as the Vigenère cipher was in the nineteenth century, *it really is absolutely secure.* [9]

A traditional one-time pad is a purely random series of characters that is possessed only by the sender and receiver. As its name indicates, the pad is used only once and then discarded. As a result, only those who have the pad can decrypt a message. In this thesis, the source text acts as a one-time pad, in that the probability of two different texts generating the same grammar is virtually non-existent.

2.5.2 Existing Mimicry Algorithms

Although existing mimicry algorithms are limited in their ability to hide data effectively, it is important to understand what alternatives are available in order to formulate an

improved algorithm. The algorithms for mimicry-based steganography fall into two categories:

- Those that use static grammars to mimic a source.
- Those that dynamically mimic a source.

These two types of mimicry algorithms are described in detail in the following sections.

2.6 Mimicry Algorithms that Use Static Grammars

The algorithms described in this section use static grammars to form their output. These grammars are referred to as “static” because they must be designed before the algorithm can be used. When using static grammars, changes in the algorithms require a substantial amount of user effort.

2.6.1 Context Free Grammars

In Chapter 7 of *Disappearing Cryptography*, Peter Wayner describes a method for mimicry that is based on Noam Chomsky’s context free grammars. Such grammars can best be illustrated by example:

Start → **noun verb**
noun → Fred || Barney
verb → went fishing **where** || went bowling **where**
where → in **direction** Iowa. || in **direction** Minnesota.
direction → northern || southern

Figure 2.4: Example of a context free grammar [11]

In this example, items in **boldface** are variables. Whenever a variable is encountered, it is replaced by a “production” of variables or phrases. In such a system, bits are encoded by the choice of phrase used to replace a variable. For example, in the sentence “Barney went bowling in southern Iowa”, the bits ‘1101’ are hidden as follows:

1. Begin with **Start** → **noun verb**

2. Expand the first variable, **noun**, as **noun** → Barney. This choice hides the bit ‘1’, as there was a choice between “Fred” and “Barney” and Barney was the second of the choices.
3. Expand **verb** as **verb** → went bowling **where** (hiding ‘1’).
4. Expand “went bowling **where**” as “went bowling in **direction** Iowa”, (hiding ‘0’).
5. Expand “went bowling in **direction** Iowa” as “went bowling in southern Iowa”, (hiding ‘1’).

This approach to text-based mimicry has its advantages and disadvantages. The context free grammar approach always produces excellent results — it is always grammatically correct, as the grammar of the output can only be that which the user defines. As well, this method has the benefit of being computationally quite simple. These benefits, however, do not outweigh the limitations inherent in the algorithm. As with any algorithm which uses a predesigned grammar, a great amount of resources (both time and effort) must be invested into designing a ‘good’ grammar. As well, once this grammar is compromised (i.e. someone has figured out how the information is being hidden), making small changes in the grammar will not suffice to restore security. In most cases, the majority of the grammar will have to be discarded.

2.6.2 NICETEXT

A later development in text-based mimicry is NICETEXT, a mimicry method proposed by Mark Chapman [4]. NICETEXT is an improvement over the ‘pure’ context free grammar method as it provides a more flexible framework for grammar based mimicry.

The NICETEXT method differs from context free grammars in that it is not limited to a single static grammar. The first step in the NICETEXT method is classification. Each word that NICETEXT recognizes is stored in a dictionary file, and is classified in a group. Words may be classified by part of speech, or by more specific groups (such as “male name” or “direction”).

Grammars are then defined in one of two ways. The first way to generate a grammar under NICETEXT is similar to the use of single variables in the traditional context

free grammar method. A grammar is defined by a list of word classifications (i.e. “name action_verb adverb”), which is more commonly referred to in the NICETEXT proposal as a ‘style’. This particular style may generate text such as “Alice runs quickly” or “Bob reads often”. Data can be hidden in the choice of each of the words by associating a sequence of bits with each word. In this particular example, if there are eight choices for “name”, three bits can be encoded each time a name is output.

The more innovative method for generating mimicked text by NICETEXT is the dynamic recognition of a grammar in a source text. A given piece of source text is parsed, and the dictionary (which must have been originally defined by the user) is used to classify each word in the source text according to the dictionary entries. The style is then used to produce output as in the first method, as described above.

The dynamic style recognition of NICETEXT allows for partial automation in the mimicry algorithm. However, NICETEXT suffers from several design flaws:

- The NICETEXT engine cannot recognize words that it has not previously encountered when training on a source text for styles. When this occurs, the user must either classify the word or accept that it will fall into a ‘random’ category.
- The initial classification process is a long and difficult one. While it may be possible to classify many words in standard English, each new field-specific jargon will require its own classification.
- Problems occur with words that have multiple meanings in different contexts (particularly in the English language). Such a problem is described in [3]:

If we look at the sentence, “It took me a long time to complete the project” then we can replace “time” with “duration”, “period” or “span” without changing the meaning of the sentence . . . Now consider another example with the sentence . . . “I’m going to *time* your next lap.” If we attempt to replace “time” with the above synonyms, we get the following undesirable results:

I’m going to *time* your next lap.
I’m going to *duration* your next lap.
I’m going to *period* your next lap.
I’m going to *span* your next lap.

While NICETEXT does improve upon the use of static grammars by incorporating a learning process, the fact remains that the very presence of a user-defined grammar limits the possibilities for outputs. In order to have a truly automated mimicry engine, the process behind the mimicry algorithm must be fully dynamic.

2.7 `mimic`: A Dynamic Mimicry Algorithm

In Chapter 6 of *Disappearing Cryptography*, Peter Wayner proposes a mimicry algorithm that works in “an automatic way”. The automatic nature of Wayner’s `mimic` algorithm inspired this project, as the algorithm suggests a more efficient and user-friendly way of hiding information within plain text (especially when compared to the mimicry that follow a set of static rules as prescribed by the user). Wayner describes his method as “an automatic way of taking small, innocuous bits of data and embellishing them with deep, embroidered details until the result mimics something completely different” [11]. He refers to this algorithm as an n^{th} -order `mimic` function. The algorithm works by “training” on a source text. In other words, the program analyzes the source text (in an automated fashion) and uses the information learned from the source text to generate the mimicked output. This novel feature is what defines the `mimic` function as a dynamic mimicry algorithm. Wayner’s `mimic` algorithm trains on a source text, S , and generates an output text, T , as follows:

In such a system, the order of the mimicry (n) determines the complexity of the statistical model that is built via frequency analysis. For example, a first-order mimicry simply generates random characters according to their statistical distribution. In a second-order mimicry, each pair of characters in the output text had to originally appear in the source text, and so on for increasing order.

Here are some examples of the kind of output generated by the `mimic` algorithm:

- First-Order:
islhne[hry saeeooisnre uo w nala al coehhs pebl e to agboean ce ed csh-
cenapch nt sibPah ea m n [tmsteoia lahid egndl y et r yf arleo awe l
eo rttntnnhtohwiseoa a dri 6oc7teit2t lenefe clktoi l mlte r ces. woeiL
, misetemd2np eap haled&oolrcc yttr tr,oh en mi elarlbeo tyNunt . syf
es2 nrrpmdo,0 reet dadwn dysg te.ewn1ca-ht eitxrni ntoos xt eCc oh sao
vhs0mhgr
- Second-Order:

1. Construct a list of all different combinations of n letters that occur in S and keep track of how many times each of these occurs in S .
2. Choose one at random to be a seed. This will be the first n letters of T .
3. Repeat this loop until enough text is generated:
 - (a) Take the last $n - 1$ letters of T .
 - (b) Search through the statistical table and find all combinations of letters that begin with these $n - 1$ letters.
 - (c) The last letter of these combinations form the set of possible choices for the next letter to be added to T .
 - (d) Choose among these letters and use the frequency of their occurrences in S to weight your choice.
 - (e) Add it to T .

Figure 2.5: Peter Wayner's `mimic` algorithm [11]

Thy etheren ante esthe ales. icone thers the ase omsictorm s iom. wac-
 tere cut le ce s mo be t Me. Y whes ine odofuion os thore cctherg om
 tt s d Thm & tthamben tin ssthe, co westitit odecra fugon tucod. liny
 Eangem o wen il ea bionBulivethe ton othanstoct itaple

- Third-Order:
 ith eas a tan t genstructin ing butionsmage ruct secate expachat thap-res
 Miamproxis is of is a to af st. This there is monst cone usectuabloodes
 it aluenget-tecte por be the andtaly com Bevers gor the Hufferess. M
 B G achasion the coduch occomprence mon Quited the ch like bitheres.
 The
- Fourth-Order:
 captionary. Image and to compression lest constance tree. Family for
 into be mode of bytes in algorith a file of that cosition algorithm that
 word even that a size summarge factal size are: ite position scien Raps.
 The is are up much length ence, the if the a refsec-ent sec-ent of fits to
 the crans usuall numberse compression A good ways that in algorith.
 The brase two wants to hidea of English Cash the are compres then
 matimes for-matimes from the data finding pairst. This only be res-sion

The algorithm described above is useful for generating text that mimics a source in a basic manner, but it does not explain how data can be hidden in the process. Wayner

elaborates in a later section of Chapter 6, describing that this task can be accomplished by “running Huffman compression in reverse”.

2.7.1 Huffman Codes

Huffman codes are named after their inventor, David Huffman (1926-1999). The Huffman code came into being in a famous and amusing manner. Given the option to either write his final exam in a graduate course, or write a term paper to “find the most efficient method of representing letters, numbers, or other symbols using a binary code”. Huffman chose the latter, and so the Huffman code was born. [10]

In order to understand how Huffman codes can be used in a mimicry engine to hide data, it is important to understand what Huffman codes are within a larger scope.

Source Encoding

Huffman codes belong to a class known as “source codes” which have their origin in the field of data communications. Frequently, before a signal is transmitted over a communications channel, a process known as source encoding is performed which substitutes codewords for source symbols to remove redundancies from the input signal. This process is also referred to as data compression, as it reduces the average codeword length \bar{L} in the transmitted code. The Huffman code is an example of a special class of source codes known as prefix codes.

Prefix Codes

Prefix codes belong to a larger class of source codes known as “uniquely decodable” codes. All uniquely decodable codes must follow the Kraft inequality:

$$\sum_{i=1}^M 2^{-l_i} \leq 1$$

where l_i is the length of codeword i .

Figure 2.6: Kraft inequality equation

The Kraft inequality can be explained as a limit that is imposed on the number of unique short codewords that can be used in any given source code.

Prefix codes possess the special property that no codeword is the prefix for another codeword in the set, making them instantaneously decodable.

Optimal Codes

The Huffman code is a special member of the set of prefix codes since it is an optimal source code. In order to understand what it means for a source code to be ‘optimal’, it is important to understand the concept of entropy.

Entropy

The term entropy was defined by Claude Shannon in his work on information theory as the fundamental limit of lossless data compression. [5]

$$H(x) \equiv \sum_{x \in S_x} p(x) \log_2 \left(\frac{1}{p(x)} \right)$$

where $H(x)$ is the entropy of the set of sources, S_x is a source symbol from the set S , and $p(x)$ is the probability of symbol S_x occurring.

Figure 2.7: Definition of Entropy

The entropy is a quantified representation of the information represented in a set of source symbols. According to Shannon’s theory, no source code can achieve a bit rate (average number of bits per symbol) lower than the limit dictated by entropy without losing information.

2.7.2 Optimal Codes

Having defined the concept of entropy, it becomes clear that any code whose average codeword length \bar{L} which approaches the limit of entropy in its lossless compression must be an optimal code. The Huffman code is one of these optimal codes, and is thus an ideal code for use in a system that wishes to encode a maximum amount of information in a limited set of symbols.

2.7.3 Huffman Coding Algorithm

The following algorithm is used to generate the Huffman coding tree [5]:

1. Create a Huffman tree node for each symbols, encapsulating a symbol by a (name, probability) pair.
2. List the nodes in order of increasing probability of their symbols.
3. Take the nodes for the two symbols with the lowest probabilities, and make them the children of a new “supernode” with probability equal to the sum of the two original probabilities.
4. Remove the nodes corresponding to the two original symbols from the list, and place the supernode in the list, in the proper position according to the combined probability of its children.
5. Repeat steps 2 and 3 until only one node remains in the list.
6. Having completed these steps, we now have a tree.
7. Decide on a label (either '0' or '1') for each direction (i.e. up/down, or left/right, depending on how the tree is drawn) in which branches can be traversed.
8. The codeword for each original source symbol is found by tracing the sequence of 0's and 1's from the root of the tree down to the leaf node of that symbol.

An example of a Huffman coding tree follows. Given the letters and frequencies of occurrence listed in Table 2.1, the Huffman coding tree in Figure 2.8 is generated:

Symbol	Frequency of Occurrence
a	1
e	2
f	1
j	1
r	1

Table 2.1: Frequency distribution of symbols used in Figure 2.8

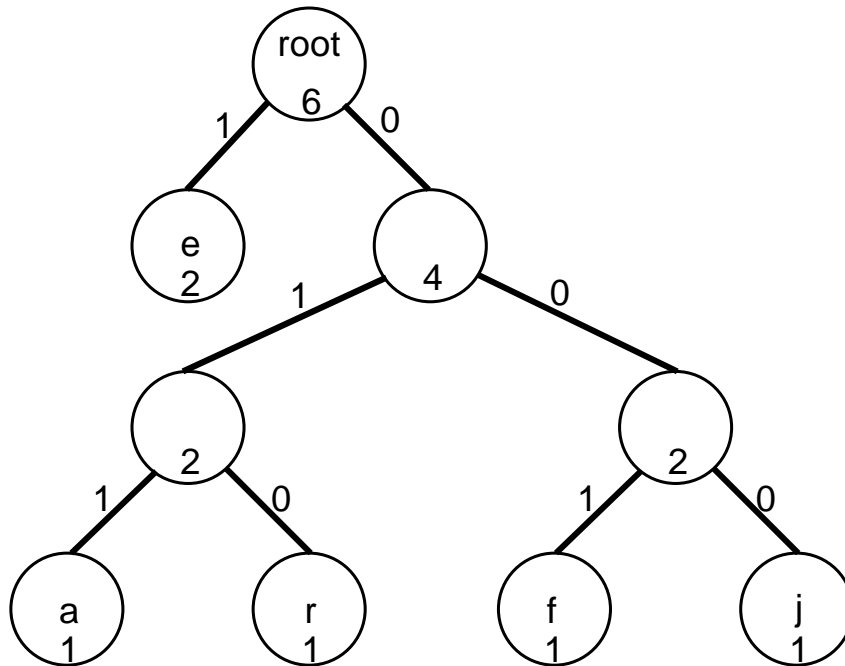


Figure 2.8: Example of Huffman coding tree corresponding to distribution in Table 2.1

2.8 Using Reverse Huffman Codes to Hide Data

By using the frequency model that is constructed by the `mimic` algorithm, Huffman trees can be generated for each choice of “next letter”. These trees can then be used to hide a stream of bits (0’s and 1’s) within a mimicked text.

Within a given Huffman tree structure, each left or right branch of a tree is assigned a value (1 or 0) (see Figure 2.8). In order to generate text (and thus hide data) using Huffman trees, the following algorithm is used:

Given a bit stream B of length N , composed of bits b_i , $i = 0, \dots, N$:

1. Begin by starting at the root of the Huffman tree (the top-most entry), and set $i = 0$.
2. Perform the following loop until $i = N$:

- (a) If the current node has no children (i.e. it is a leaf node), output the label of the node.
- (b) If $b_i = 1$, set the current node to that found by descending along the left branch. Otherwise, $b_i = 0$: Set the current node to that found by descending along the right branch.
- (c) Increment i .

For example: using the Huffman tree from Figure 2.8, the following outcomes are possible:

Bit Sequence	Output
1	e
000	j
001	f
010	r
011	a

Table 2.2: Reverse Huffman Codes example

Note that the brief description of this algorithm is only intended to illustrate the basic process of traversing a Huffman tree in reverse. The full description of the algorithm implemented in this thesis can be found in Section 3.2.1.

Chapter 3

A New Algorithm for Mimicry-Based Linguistic Steganography

3.1 Formulating a New Mimicry Algorithm

The weaknesses of Wayner’s *mimic* algorithm have been described. While the algorithm is capable of generating statistically equivalent text at all orders, only at orders higher than 5 do “real” English words begin to emerge. However, even when these words are generated, there is not a continuous flow in the resulting output text. In an attempt to formulate an algorithm that produces superior output text, three specific considerations are made:

1. Using words as the atomic unit for mimicry;
2. Preserving the case of the letters in each word; and
3. Giving special consideration to punctuation characters.

3.1.1 Using Words as the Atomic Unit for Mimicry

Wayner’s algorithm uses single characters from the ASCII ¹ character set as the basic unit, or “atom” in the mimicry algorithm. Thus, in an nth-order mimicry, each group of

¹American Standard Code for Information Interchange

n subsequent characters can be found in the source text. By changing this atomic unit to that of a single word (or punctuation character; see Section 3.1.3), this thesis hopes to generate a more realistic output text that follows the “rules” of the source text’s grammar. The resulting algorithm is almost identical to Wayner’s *mimic* algorithm, but with letters replaced by words as the atomic unit. Given a source text S , an output text T will be generated by an n^{th} -order mimicry as follows:

1. Construct a list of all different combinations of n **words** that occur in S and keep track of how many times each of these occurs in S . (This process is often referred to as “training” on the source text).
2. Choose one such combination at random to be a seed. This will be the first n **words** of T .
3. Repeat this loop until enough text is generated:
 - (a) Take the last $n - 1$ **words** in T .
 - (b) Search through the statistical table and find all combinations of **words** that begin with these $n - 1$ **words**.
 - (c) The last (n^{th}) **word** in these combinations form the set of possible choices for the next **word** to be added to T .
 - (d) Choose among these **words** and use the frequency of their occurrences in S to weight your choice.
 - (e) Add **the selected word** to T .

Figure 3.1: Modified mimicry algorithm from Figure 2.5, with changes in **boldface**

3.1.2 Preserving the Case of Letters within Each Word

Another factor that affects the quality of the mimicry engine’s output is how it deals with case (i.e. lowercase/uppercase letters). Wayner suggests that case can either be preserved or ignored. If case is preserved, the resulting set contains 52 alphabetical characters (2×26) plus punctuation and whitespace. If case is ignored, the size of the set of alphabetical characters reduces to 26 capital letters. However, the resulting output text does not mimic typical English writing very well when only capitals are used.

The algorithm employed in this thesis preserves the case of each word. Hence, if two identical words that differ in their capitalization (“orange” and “Orange”, for example) appear in the original text, there will be two separate entries in the frequency table.

The decision to preserve capitalization complements the decision to treat punctuation characters as individual entities, as capital letters frequently follow certain punctuation characters (See 3.1.3).

3.1.3 Special Considerations for Punctuation Characters

Once the decision has been made to deal with words as the atomic unit for mimicry, punctuation characters are no longer automatically dealt with by the algorithm. The proposed solution to this problem is dealing with punctuation characters as individual “words” themselves. The following punctuation characters are considered in this implementation:

Name	Character	ASCII Value
Period	.	46
Comma	,	44
Semicolon	;	59
Colon	:	58
Exclamation Mark	!	33
Question Mark	?	63
Double Quotes	..	34
Parentheses	()	40, 41
Square Brackets	[]	91, 93
Braces	{}	123, 125

Table 3.1: Punctuation characters that receive special treatment in the mimicry algorithm

The intended side effect of singling out specific punctuation characters as individual word elements is the reproduction of an output text that more closely mimics the grammar of the source text. These advantages arise from the regularity of the English language’s structure:

- Capitalized words typically follow periods, exclamation marks, question marks, and often follow colons, semicolons, and quotation marks.
- Certain words are more likely to occur after specific punctuation characters. For example, words that often follow semicolons are “therefore”, “thus”, and “however”. Using punctuation as an individual unit in the training stage thus yields more realistic results in the output text.

Once the rules have been predefined for which punctuation characters receive special treatment, another intended benefit presents itself. By treating the characters as individual entities, rules for whitespace in the output text can easily be established. For example, in standard English, two spaces always follows a period, question mark, or exclamation mark, while only one space follows a comma, colon, or semicolon.

Those punctuation characters that are not dealt with as individual word elements are considered as part of their adjacent word by the mimicry algorithm. These characters are neglected as they do not correspond to a predictable grammar. Future implementations will simplify the classification of punctuation characters by the user, or make the procedure unnecessary (see Section 5.2).

3.2 Mimicry Algorithm Implementation Details

This second phase of the thesis discusses the details involved in implementing the mimicry algorithm described in previous sections.

3.2.1 Encoding: Overview

The following is a rough outline of the algorithm used to hide a secret message within stegotext, given a secret message M (of length N characters), source text S to train upon, and order n :

1. Build an n^{th} -order frequency model of the source text S by parsing it, word by word (as defined by punctuation and whitespace delimiters).
2. Convert the secret message M into a stream of bits, B consisting of bits b_i , $i = 1, 2, \dots, 8N$.

3. Use B in conjunction with the frequency model to produce the output stegotext T .

3.2.2 Hash Tables and Huffman Trees: Building the Frequency Model of the Source Text

In this thesis, the frequency model is built for an order n mimicry by constructing a tree of n nested hash tables. That is, there is one primary hash table for all possible first words; under each entry in that hash table, there is another hash table containing all possible words that follow it. This nesting proceeds to depth n , to allow the storage of the frequency statistics up to the n^{th} order.

Hash Tables

Since there is so much data derived from the source text S , it is important to store the information in an efficient data representation. Due to the infinite number of possible words, it is not possible to store the frequency data in a static structure (such as an array). One of the best possible solutions that is often used to store an arbitrary number of strings is an open hash table.

Open hash tables work in the following manner:

- An array of fixed size is allocated to hold the data. Each entry in the array contains a linked list which acts as a ‘bucket’ for all strings that map to the index of that entry.
- Each string is run through a ‘hashing function’ which generates an index to a destination bucket. If there is already a string in the bucket, a collision occurs, and the string is added to the end of the linked list in the bucket.

Perfect Hashing vs. Open Hashing Given a predetermined set of strings (i.e. in a source text that is available in its entirety), it is possible to construct a perfect hashing function that maps one string to each bucket in the hash table. When a perfect hashing function is used, no collisions occur in the hash table.

The alternative method is open hashing. In open hashing, a predefined hashing function is used. While such a system allows for collisions (which potentially increase the run

time of both hash table population and hash table lookups), open hashing is used in this thesis for its simplicity and lower memory usage.

Selecting a Hashing Function The purpose of a hashing function is to populate hash objects in an even distribution across all buckets. The ELFHash function [1, 2], commonly used for hashing strings in Linux “**E**xtended **L**inker **F**ormat” binaries, is used here as it provides an even distribution for strings of arbitrary length.

Selecting Hash Table Sizes In the first implementation of this thesis, the hash tables were made quite large to reduce the number of insertion collisions. However, when preliminary testing was performed, the functional limit on the system’s performance proved to be physical memory limitations. When memory was allocated beyond available physical memory, the system slowed down below the limits of acceptable performance. A decision was made to allocate memory for each hash table by dividing the number of entries by two in each subsequent level. This exponentially decreasing design was chosen somewhat arbitrarily, but proved to be efficient for this implementation. Since the maximum mimicry order was limited to 8 in this implementation, the “root”, or first order table, contains $2^9 = 512$ entries. The second order table contains half that number of entries (256), the third order table contains 128, and so on.

Parsing the Source Text

The following algorithm is used to parse the source text S into a large tree structure of hash tables for an order n mimicry:

1. Build a list of $n - 1$ ‘previous’ pointers, all initially set to NULL. At any given time during the parsing procedure, there will be $n - 1$ sequences of words being maintained. The longest sequence will be composed of the $n - 1$ words $W_{i-n+1}, \dots, W_{i-1}$ preceding W_i , and the shortest sequence will consist only of W_{i-1} . These pointers refer to the last words in each of these sequences.
2. For each new word W_i encountered,
 - (a) Check to see if W_i is in the root hash table. If not, add it and set its “number of occurrences” $n_o(i)$ to 1. If it is there, increment $n_o(i)$ by 1.

- (b) For each of the ‘previous’ pointers, check to see if W_i is in the list of possible next words. If not, add it and set its n_o to 1. If it is there, increment n_o by 1.
- (c) For each sequence $j = 1 \dots n - 1$, update the pointer for that sequence so that the last word in the sequence refers to the W_0 .
- (d) Update pointer 0 so that it refers to the entry added to the root table.

At the end of this procedure, a tree of hash tables containing the structure and occurrence of words in the source exists. However, in order to use this structure to hide data, the information must be subsequently converted into a Huffman tree representation.

Generating the Huffman Coding Trees

For each hash table and each of its descendants, a Huffman coding tree is generated. This procedure is performed according to the Huffman coding tree algorithm in Section 2.7.3. Two differences to note are:

1. No Huffman tree is generated for the root table, since the selection of words from the root table is random.
2. As the Huffman tree is generated for each hash table, the codeword that is generated by the process is stored in each word entry to later facilitate the decoding process.

The following is the Huffman tree corresponding to the root table entry for the word “a”. It was generated by training upon the source text found in Appendix A, Figure A.1, and reflects the data that can be encoded by the possible choices of next word (“trio”=1, “hoax”=011, “rich”=010, “rabbi”=001, “rumor”=000).

3.2.3 Converting Characters to Bits

In order to hide bits of data within a mimicked text, the stream of characters in the secret message must first be converted into a stream of bits. In this thesis, this conversion is performed in a “little-endian” manner, storing the least significant bit first. The value of the first bit (0 or 1) is obtained by performing the bitwise AND operation (&). That is, for character C in M , bit 0 ($b_0(C)$) can be determined as $b_0(C) = C \& 1$. In general, bit i of character C can be determined as:

$$b_i(C) = (C \gg i) \& 1$$

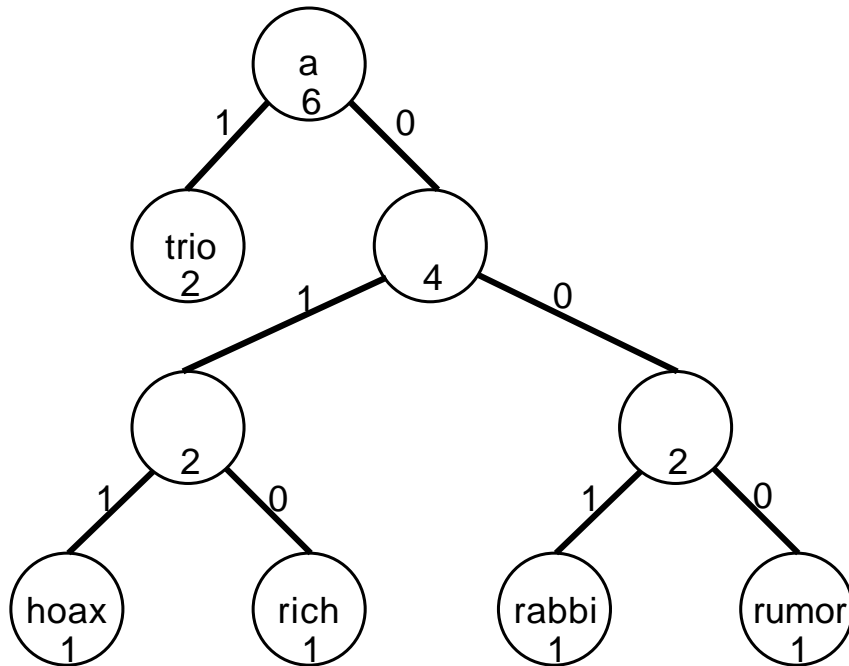


Figure 3.2: Example of Huffman coding tree using words as the atomic unit

where \gg is the “logical bitshift right” operator. By performing this operation for each character C in M , the bit stream B is generated.

3.2.4 Producing the Stegotext

Once the frequency distribution for the source text S has been generated (with accompanying Huffman trees), and the secret message M has been converted into the stream of bits, B , the stegotext generation process is simple:

1. Begin by selecting a random word in the root table: call this word W .
2. If the root of the Huffman tree for W has no child leaf nodes, no other word can follow W . Pick a new random word in the root table and proceed, using it as the root node of the Huffman tree.
3. Otherwise, the root of the Huffman tree for W contains at least one child leaf node. If there is only one child node, output the word W' corresponding to the child node, and use the node for W' as the next root node.

4. If the root of the Huffman tree for W has at least two child leaf nodes, use each bit b_i in the bitstream B to descend along the branches of the Huffman tree until a leaf node is reached corresponding to word W'' .
5. When a leaf node is reached, output word W'' (as part of the stegotext), and set $W = W'$.
6. Loop to step 2, repeating as long as there are bits to be encoded.

After this process is complete, all bits b_i that are generated from the secret message M will be encoded in the resulting stegotext T .

Sample Output

The following texts are portions of sample outputs generated by the algorithm presented within this thesis from the source text `Lab3.txt`. First-order texts are not included as first-order mimicry is not implemented; this is simply due to the fact that first order texts are a random collection of words from the source text.

- Second-Order:
zero-valued or have very small values: most of the reconstructed image. The values. The case. The case. The basic idea behind the iterative method rather than the direct method that it is used in magnitude) were actually reconstructed image, it is being transmitted across a program was performed the iterative approach can be larger than all four sub-bands contain much smaller(i. e. m(in magnitude(in magnitude to use the compaction and 5 of the input image was found in
- Third-Order:
This algorithm is iterated upon until there is only one average coefficient(in magnitude) coefficients. The sparseness ratio, S , was provided. We were then asked to write a program was written that can be found in the second two elements. The wavelet coefficients, the method which is used in the denominator of $\sqrt{2}$. This 2-D Haar Transform was performed again on the LL subband(and so on).
- Fourth-Order:
Both the direct and iterative methods of the Haar 2-D Transform are quite efficient; however, the method which is used should be chosen judiciously according to the application. normalization condition imposed by taking averages and differences. This is useful for compression if the differences are small enough, they can essentially be discarded, leaving a" fuzzy" representation of the source image. The procedure was

performed again on the image I2, yielding similar results. The largest coefficient was found in the top-left coefficient with a value of 101.

- Fifth-Order:

Wrapper code for the 2-D Haar Transform section of the assignment is contained in Part2. m(B. 6). This 2-D Haar transform was performed on images I1 and I2. The Haar transforms(IH1 and IH2) of I1 and I2 are: The images were then reconstructed by preserving 32, 13, or 5 coefficients, the subjective(i. e. via run length coding). When the image is reconstructed, the reconstruction is performed using 32, 13, and 5 of the largest coefficients. This procedure resulted in the following sparseness ratios: Reconstructed images can be seen in the iterative method.

3.2.5 Decoding: Overview

The decoding process is analogous to running the encoding process in reverse. Given a source text S , stegotext T and order n ,

1. Build an n^{th} -order frequency model of the source text S by parsing it, word by word.
2. Use T in conjunction with the frequency model to reconstruct the bitstream of the secret message B .
3. Convert the bitstream B back into the secret text message M .

The process for building the frequency distribution was discussed in Section 3.2.2.

3.2.6 Using the Frequency Model to Reconstruct the Stream of Bits from the Stegotext

In order to reconstruct a stream of bits from the stegotext T (containing N words), the same source S and order n that were used to construct the stegotext must be specified. Once these two parameters are present, the identical frequency distribution model can be constructed and decoding can occur according to the following procedure:

1. Begin by looking up W_0 in the root table. If it is not present, then the source texts must differ and decoding has failed.

2. For all subsequent words in the stegotext, check to see if the current word, W_i , is in the Huffman tree H for possible words that can follow words $W_{i-n+1}, \dots, W_{i-1}$ (the previous $n - 1$ words).
3. If W_i is present in H , output the sequence of bits corresponding to the selection of W_i (i.e. the Huffman codeword), and update the list of previous words to be W_{i-n+2}, \dots, W_i .
4. If W_i is not present in H , begin the algorithm again by looking up W_i in the root table and setting the list of previous words to be W_i .

At the end of this procedure, the output will consist of a stream of bits corresponding to a sequence of ASCII characters. These bits must be converted into their corresponding characters to reproduce the secret message M .

3.2.7 Converting Bits to Characters

The process for reconstructing characters from bits is a simple one, and requires only two bitwise operators: OR ($|$) and logical shift left ($<<$). The algorithm operates as follows:

1. Initialize an empty character $C = 0$.
2. Eight bits are used to reconstruct a single character; as bit i is received, $0 \leq i \leq 7$, set $C = C | (1 << i)$.

For example, the bit stream “00101110 00010110 10100110”² would be parsed into the three characters ‘t’ (01110100), ‘h’ (01101000), and ‘e’ (01100101), forming the word “the”.

²Spaces have been inserted for clarity to illustrate 8 bit boundaries between characters

Chapter 4

Analysis

4.1 Subjective Analysis

While one goal of mimicry is to defeat detection by automatic computer-based systems, the true test of the mimicry algorithm's success is how well the mimicry performs when a live person is reading the stegotext. In order to evaluate the success of the mimicry algorithm in terms of human perception, a survey was designed and distributed to a small number of participants (Appendix B). This survey asks questions of the participant about a set of texts generated by the mimicry algorithm. The survey addresses technical questions of logical consistency, verbal flow, and grammatical errors in the text. It goes on to ask the reader if the text resembles the style of the original author (if they are familiar with the author's work) and even proceeds to ask if the reader suspects that a computer has generated the text.

4.1.1 Selection of Test Cases

In the survey, six pieces of stegotext were generated, all hiding the same short message. A second, fourth, and sixth order text that mimicked Shakespeare's epic play *Macbeth* were generated, and stegotexts of the same order were created to mimic a lab report on the Haar wavelet transform (written by the author of this thesis).

The texts were presented by being grouped by source (i.e. texts generated from *Macbeth* appeared together, as did texts generated from the lab report). The sequence in which the different order texts appeared was randomized once and the resulting sequence was

presented to all surveyed.

4.1.2 Results

While the scope of this project did not allow for an extensive subjective survey, preliminary results yield positive indications for the success of the mimicry algorithm. Analyzing the overall response suggested the following, verifying some expected results:

- Logical flow, verbal flow, and resemblance to the author’s work increase with increasing order n .
- Punctuation errors and grammatical errors decrease with increasing order n .
- Overall results are more successful when the source follows a more structured format (i.e. the lab report was more successful than Macbeth). In these cases, logical and verbal flow are ranked higher, and punctuation and grammatical errors are fewer in number.

It is also interesting to note that the text with which the respondents were less familiar (the lab report) was hardly ever suspected to be computer generated; in contrast, the text with which all of the readers were familiar (Macbeth) only appeared to be ‘authentic’ (i.e. not computer generated) at the highest order of mimicry.

4.2 Frequency Analysis

One essential method for evaluating the strength of a mimicry-based steganography algorithm is frequency analysis of the generated text. The algorithm presented within this thesis, in theory, produces a statistically equivalent text to the source text S . This section attempts to verify the statistical equivalence (by frequency distribution of characters) between source texts and generated stegotexts. While this statistical equivalence does not verify the correct mimicry of the source text, it does demonstrate that the requirement of matching the statistical distribution of the source text is still satisfied when words are used as the atom in the mimicry algorithm.

4.2.1 Selection of Test Cases

For each of the following tests, four source texts were used: ¹:

Filename	File size (bytes)	Description
Lab3.txt	10307	A fourth year lab report on the Haar wavelet transform (written by the author of this thesis).
chapter5.txt	27632	A portion of chapter 5 of Peter Wayner's book, <i>Disappearing Cryptography</i> .
macbeth.txt	100321	William Shakespeare's epic play, <i>Macbeth</i> .
inferno.txt	684987	The Inferno: A book from <i>The Divine Comedy of Dante Alighieri</i> , translated by Henry Wadsworth Longfellow.

Table 4.1: Files used as source texts in analysis of the mimicry algorithm

In order to estimate the performance of the mimicry algorithm for all allowed orders, tests were performed on each of the four source texts for orders 2 through 8. For each source text S and order n , a stegotext T was generated based on the secret message “This is a secret message.” (a 26 byte message consisting of 20 alphabetical characters, four spaces, a period and a single newline “return” character (ASCII value 10)).

4.2.2 Results

Figures 4.1, 4.2, 4.3, and 4.4 show the percentage of characters found for each character in the alphabet in each resulting text. The leftmost bar in each plot represents the frequency of occurrence of the letter in the source text, and the subsequent 7 bars reflect the frequency of occurrence in mimicked stegotext of increasing order.

¹Shakespeare's *Macbeth* and Dante's *Inferno* are public domain and were downloaded from Project Gutenberg at <http://promo.net/pg>

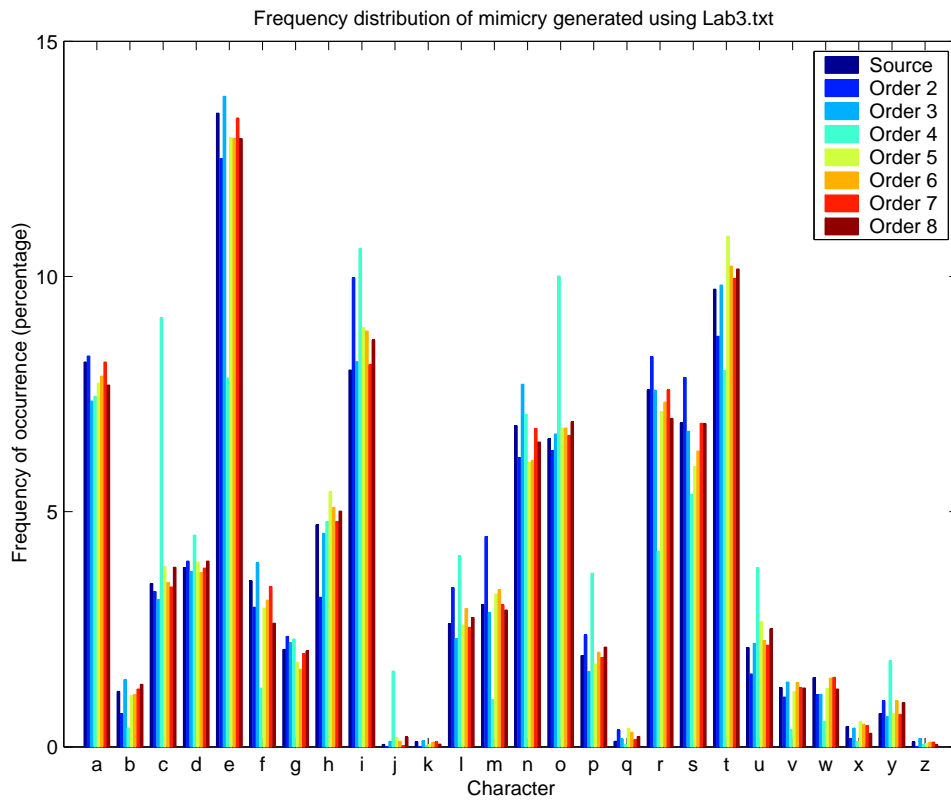


Figure 4.1: Frequency distribution of stegotexts generated using Lab3.txt as the source text

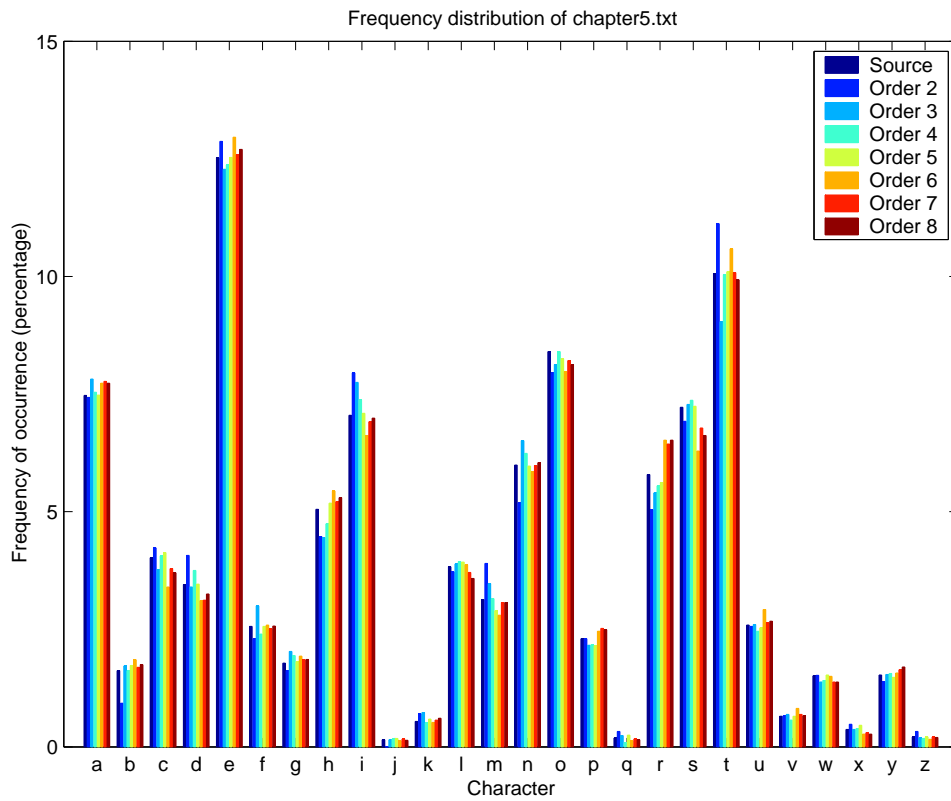


Figure 4.2: Frequency distribution of stegotexts generated using `chapter5.txt` as the source text

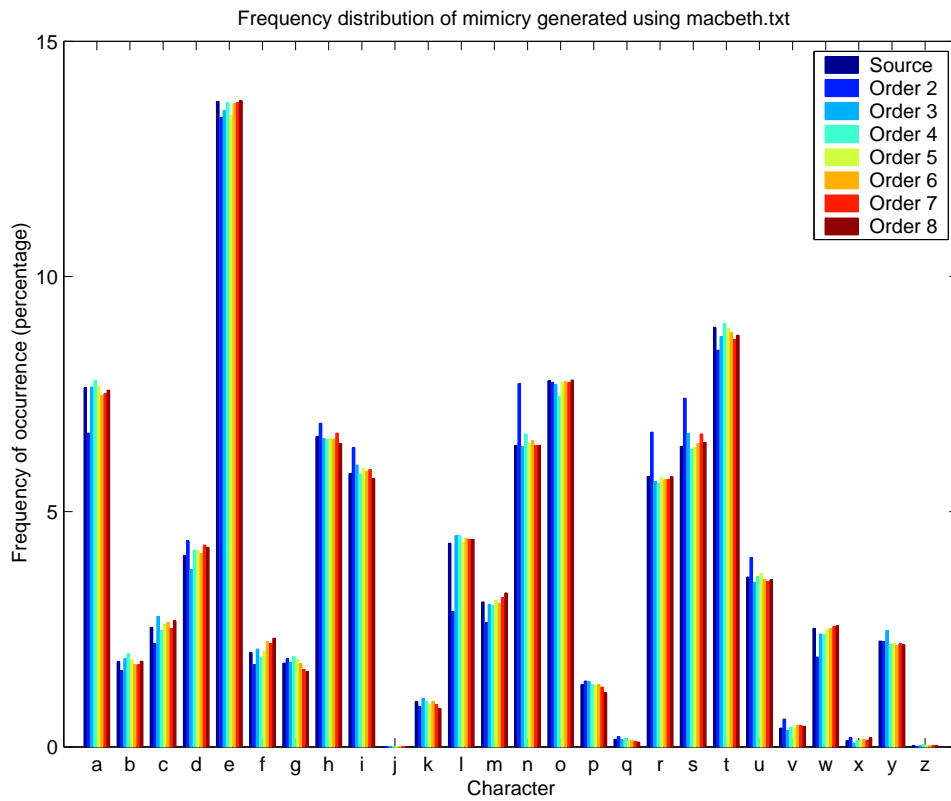


Figure 4.3: Frequency distribution of stegotexts generated using `macbeth.txt` as the source text

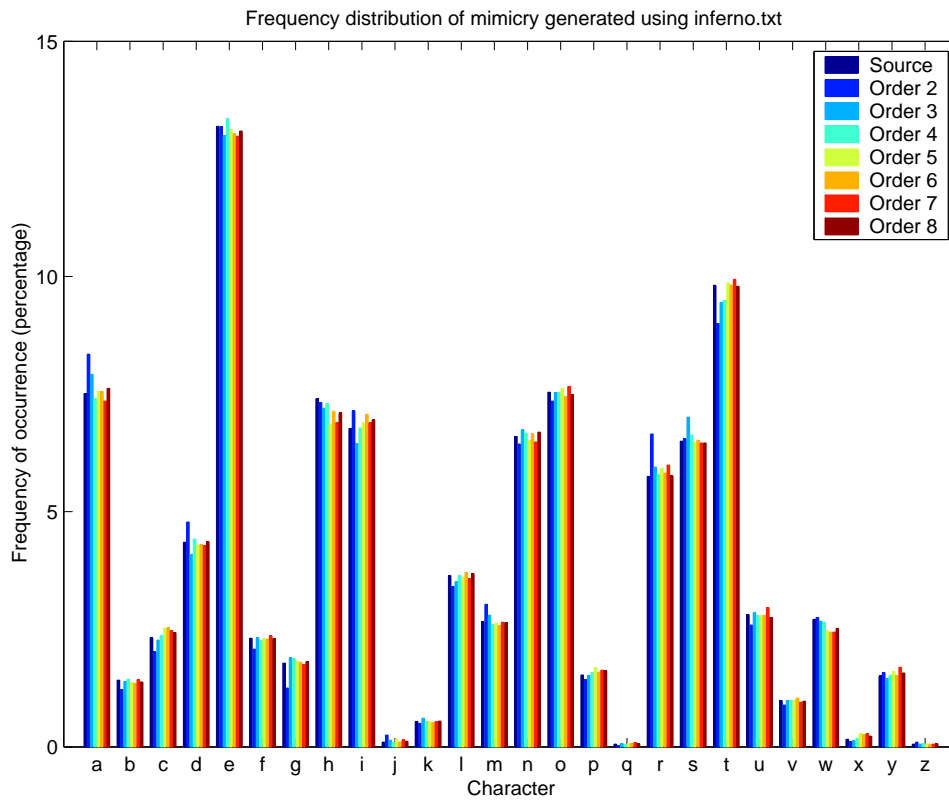


Figure 4.4: Frequency distribution of stegotexts generated using `inferno.txt` as the source text

These figures indicate that the output text T almost always mimics the frequency distribution of the source. Some large irregularities occur at lower orders (of particular interest is the strange distribution in the fourth order run in Figure 4.1). Such irregularities appear to be source text specific — they appear to occur when there are loops in the output text (i.e. repeating certain n word clauses), which only occurs in shorter source texts for specific orders. While irregularities do occur, the trend suggests that the frequency distribution of the stegotext approaches that of the source text with increasing order.

4.3 Testing Various Parameters of the Mimicry Engine

While the focus of this thesis is an examination of the strength of the mimicry algorithm, the practicality of employing the algorithm in everyday use must also be investigated. The following sections describe various tests that were performed on different parameters of the mimicry engine and their results.

4.3.1 Expansion Factor vs. Order

One of the tradeoffs that is made when using a mimicry-based steganography system rather than a traditional one is the “bloating” of the resulting stegotext. This mimicry overhead can be represented in an analogous form to that of data compression. The term “expansion factor” (E) can be used, and is defined as:

$$E \equiv \frac{\text{Size of stegotext (bytes)}}{\text{Size of secret message (bytes)}}$$

The first set of tests investigate the effect of changing the order of the mimicry on the resulting expansion factor.

Selection of Test Cases

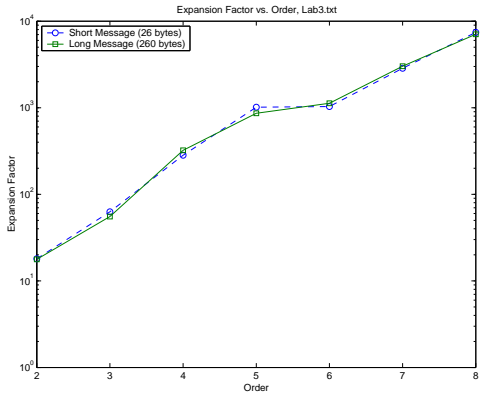
In each of the tests relating order to expansion factor, two secret messages were used. The first message used was “This is a secret message.”. The second message was simply the first message, repeated ten times, yielding a 260 byte message. Even though the effects of secret message length were not being tested for explicitly, the two secret

messages of different lengths were used to examine whether or not the trends were invariant with respect to secret message length. Section 4.3.2 goes into further detail regarding the effects of secret message length on expansion factor.

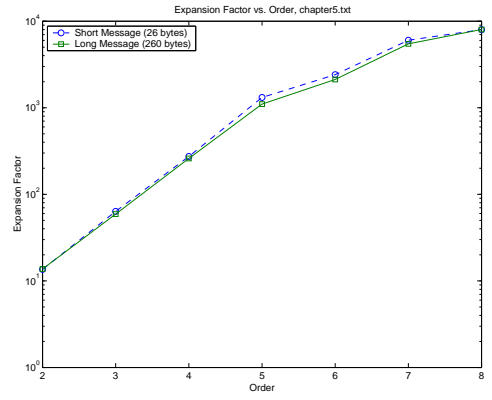
For each of the four source texts, mimicry was performed on both of the secret messages for all orders between 2 to 8. Each call to the mimicry algorithm was repeated 100 times to ensure a valid statistical sample, and the average value of the expansion factor E from each of these sets was recorded.

Results

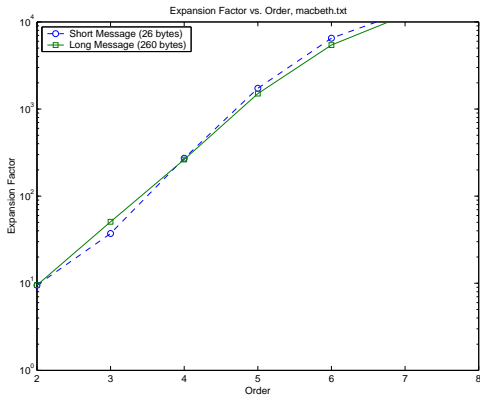
Figures 4.5(a), 4.5(b), 4.5(c), and 4.5(d) represent the data collected from each test case, and are shown on the following page.



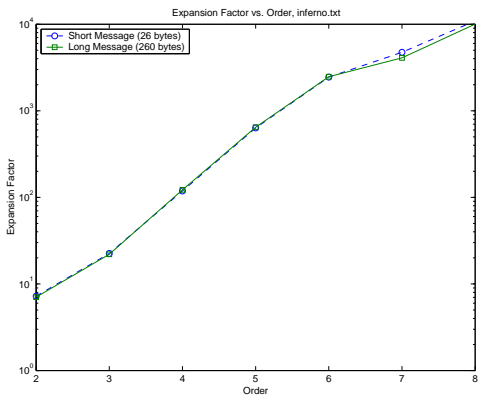
(a) Lab3.txt



(b) chapter5.txt



(c) macbeth.txt



(d) inferno.txt

Figure 4.5: Testing the effect of changing order on expansion factor E

In each case, the expansion factor E is plotted against the mimicry order n , using a logarithmic scale for E . The results indicate a trend: expansion factor grows exponentially with increasing order. This exponential growth is identical for the two input messages of different lengths.

One point of interest is how the curve seems to flatten out at higher orders. While it was not possible to investigate the behaviour of the mimicry algorithm at much higher orders (or desired, as the exponential growth of the expansion factor makes use of the mimicry algorithm at very high orders quite impractical), the plateau indicates an asymptotic tendency of the algorithm. A plausible explanation for this behaviour follows from the nature of the mimicry algorithm at high orders:

When the mimicry algorithm constructs its frequency model of the source text, it does so based upon the order n . To generate the output text/stegotext, a random start point is selected and the Huffman trees are traversed. As n approaches and passes the average number of words in a repetitive sequence (approximately 7-10 words), the number of possible choices near the end of such a sequence tends to zero. When there is no possible choice for the next word, a new random starting point is selected. As such, the output consists of a sequence of random sentences of the maximum sequence length (even as n becomes greater than this value). This could explain the apparent plateau in the preceding graphs.

4.3.2 Expansion Factor vs. Message Length

Another interesting relationship that was analyzed is the effect of changing the secret message length on the expansion factor E .

Selection of Test Cases

For this series of tests, only one source text was used (`chapter5.txt`). However, the secret message (and repeated versions of it) “This is a secret message.” from the previous section was reused. The following secret message files were generated:

These six secret message files were used with the mimicry algorithm for orders between 2 and 5 (higher orders could not be mimicked due to limitations on available disk space). Once again, each call to the mimicry algorithm was repeated 100 times to provide a

Filename	File size (bytes)
testMessage1	26
testMessage10	260
testMessage100	2600
testMessage1000	26000
testMessage10000	260000
testMessage100000	2600000

Table 4.2: Secret message files and their corresponding sizes, used to test the effect of changing secret message length on the expansion factor E

statistically significant set of samples.

Results

The following graph represents the data collected from the aforementioned tests. Linear fits on the collected data suggest that there is a directly proportional relationship between the input message length and stegotext (output) size. The slope of the resulting linear fit is identical, regardless of input message length.

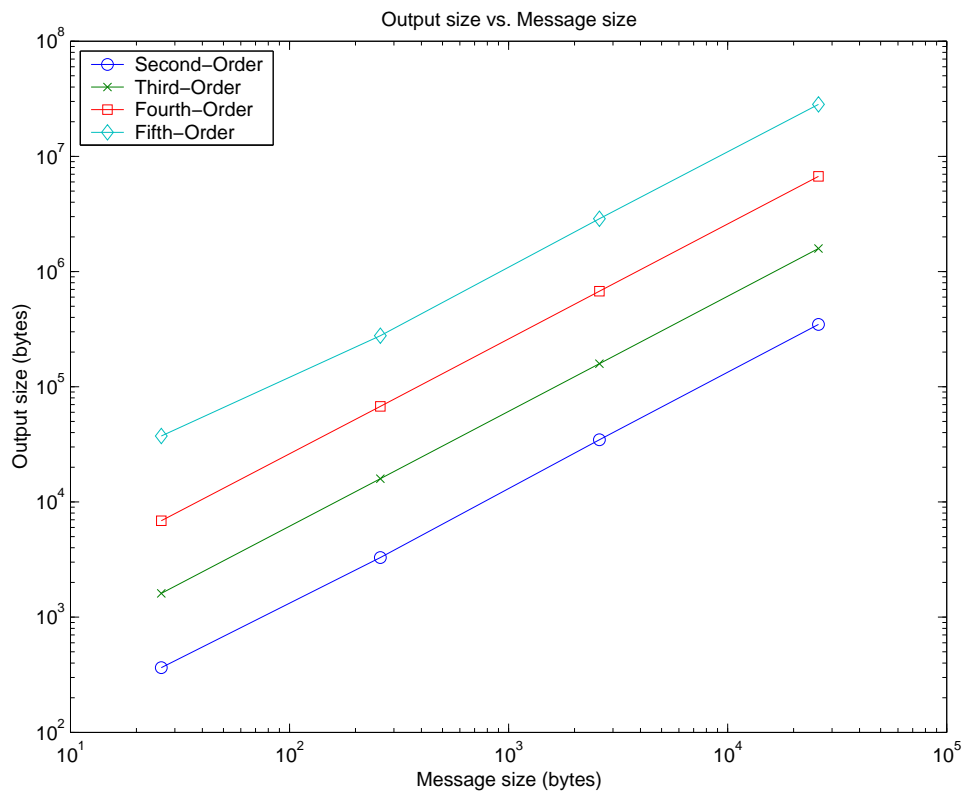


Figure 4.6: Testing the effect of changing secret message length on expansion factor E

4.3.3 Timing Tests

The standard benchmark for testing the practicality of an algorithm for everyday use is the timing test — timing how long the algorithm takes to run. Ideally, testing should be performed on various systems under different operating systems and configurations. Since neither various test platforms nor the time for such extensive testing were available for this thesis, some simple tests were run to estimate the effects of different source text size and order on run time.

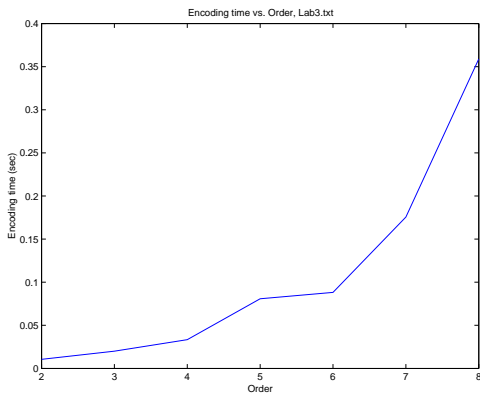
The test platform for all runs of the mimicry algorithm was an AMD Athlon (600 MHz) processor running Redhat Linux 7.1, Kernel 2.4.7-10 with 256 MB of RAM.

Selection of Test Cases

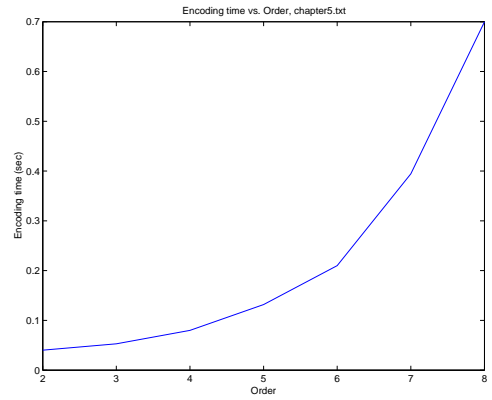
Testing was performed on each of the four available source texts, for all orders between 2 to 8, using the standard test message “This is a secret message”. Each test was run 50 times to correct for possible variance due to CPU load, and results were averaged to produce the following results.

Results

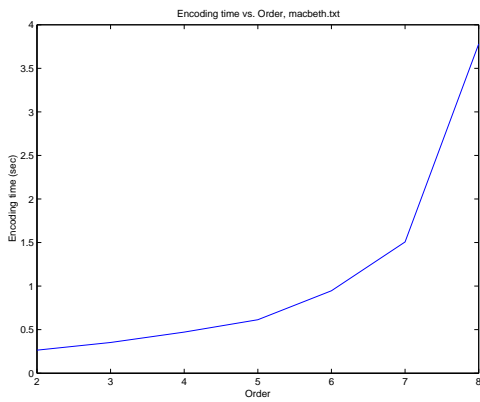
Figures 4.7(a), 4.7(b), 4.7(c), and 4.7(d) are shown on the following page.



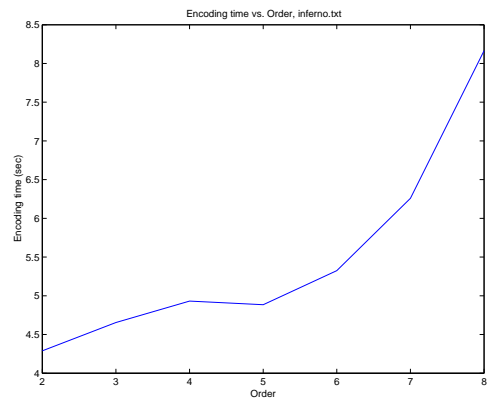
(a) Lab3.txt



(b) chapter5.txt



(c) macbeth.txt



(d) inferno.txt

Figure 4.7: Mimicry timing results for various source texts

The results from this series suggest the following:

- There is a constant offset for the amount of time required to parse the source text that is related to its size.
- The amount of time required to build the frequency distribution model for a source text S is exponentially related to the mimicry order n .

These results demonstrate that even on a workstation with modest computational power, implementation and use of the mimicry engine for linguistic steganography is quite feasible, even at higher orders.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

All of the results gathered suggest that this thesis is successful in developing an automated, dynamic algorithm to perform data hiding and data recovery using mimicry-based linguistic steganography. The algorithm presented within is developed using Peter Wayner's `mimic` algorithm as a starting point. The main change from Wayner's algorithm is the usage of words (and punctuation characters) as the atomic unit in mimicry rather than characters. This change causes the source text training to not only mimic the statistical distribution of the source text, but also to mimic the grammatical structure found within. However, in contrast to mimicry engines that use static grammars, the mimicry algorithm is abstracted by limiting the atomic unit to words. This abstraction allows mimicry of the grammatical structure of the source regardless of the source style or language.

Subjective studies suggest that the algorithm succeeds in producing stegotext that is capable of 'fooling' human readers for fourth order and higher. Success in producing a believable stegotext is closely linked to the quality of the source text.

Frequency analysis suggests success of the algorithm at all orders. Every order of mimicry produces a stegotext which has a character distribution that is statistically equivalent to the source, within acceptable limits.

Finally, the algorithm employed in this thesis proves to be quite practical

when timing tests are performed on a modestly powered workstation. The expansion factor of output text is quite high for fourth order and higher, but this does not become a factor unless message texts grow large. Such large expansion factors may also improve the success rate of the mimicry algorithm as long sentences from the source are likely to appear at higher orders.

5.2 Future Research

While this thesis has explored a new method for mimicry-based linguistic steganography, the algorithm proposed within has not been studied in as much detail as would be possible in a project with larger scope. Possible areas that should be investigated are:

- A more in-depth study of the strength of the proposed algorithm. A strong mathematical model for evaluating the security of steganographic systems such as the one proposed within this thesis has not yet been developed. Once such a model is available, it should be applied to the algorithm found within in order to analyze the strength of the data hiding process.
- Grammar checking tools. As linguistic steganography becomes more popular, following the statistical distribution of the English language may no longer suffice to defeat a computer interceptor. Software tools to perform grammatical analysis on messages may be employed to detect possible stegotexts. The ability to ‘fool’ such systems would be a good test of the success of a mimicry algorithm.
- More subjective studies. As was mentioned in the Analysis chapter, the time available for this project did not allow for extensive subjective studies. A better study would have many participants, and could present both human-generated and computer-generated texts to better determine if participants could differentiate between the two.

As in any design oriented project, there is no limit as to the number of features that can be explored. The following are some possible paths that could be pursued in expanding the scope of this thesis:

- The thesis presented within is designed with the intent of being used with source texts that are written in the English language. No consideration is made for the

structure of foreign or computer based grammars (such as programming languages). The algorithm could be enhanced so that it could learn about more than the frequency distribution of the words in the source text. This enhanced algorithm would be adaptive to the punctuation and spacing of the source text, regardless of the language it was written in. Such a system would be quite fascinating, as it would be capable of mimicking (and hiding data within) **every** style of plain text: C files, systems of mathematical equations, prose, and poetry. Approaching this problem would require an excellent knowledge of computer learning.

- As previously discussed, the strength of the system proposed in this thesis is unknown. An additional measure of security could be incorporated into the system by encrypting the secret message before using it to generate the stegotext. This addition would greatly increase the strength of the overall system as the secrecy of the message would remain intact even if the steganography system was defeated. The stegotext may even mimic the source more successfully, as the reverse Huffman coding algorithm relies on a random stream of bits as its input, and the bit stream generated by an encryption algorithm is typically more random than a string of characters.

Bibliography

- [1] Andrew Binstock. Is ram speed making your hashing less efficient?, 2001. URL: <http://www.ddj.com/documents/ddj9604b/>.
- [2] Andrew Binstock and John Rex. *Practical Algorithms for Programmers*. Addison Wesley Professional, 1995.
- [3] Mark Chapman, George I. Davida, and Marc Rennhard. A practical and effective approach to large-scale automated linguistic steganography. In George I. Davida and Yair Frankel, editors, *Information Security: 4th International Conference, ISC 2001, Malaga, Spain, October 2001, Proceedings*, pages 156–165. Springer-Verlag, 2001.
- [4] Mark T. Chapman. Hiding the hidden : A software system for concealing ciphertext as innocuous text. In Yongfei Han Tatsuaki and Okamoto Sihan Qing, editors, *Information and Communications Security First International Conference, ICICS 97 Beijing, China, November 11-14 1997 Proceedings*. Springer-Verlag, 1997.
- [5] Simon Haykin. *Communication Systems*. John Wiley & Sons, Inc., 2001.
- [6] David Kahn. *The Codebreakers: The Story of Secret Writing*. Scribner, 1996.
- [7] Ellen Messmer. Government restrictions on encryption pose obstacles for internet security, 1998. URL: <http://www.cnn.com/TECH/computing/9805/19/encryption/>.
- [8] Thomas Mittelholzer. An information-theoretic approach to steganography and watermarking. In Andreas Pfitzmann, editor, *Information Hiding: Third International Workshop, IH'99, Dresden, Germany, September/October, 1999 Proceedings*, pages 1–16. Springer-Verlag, 1999.

- [9] Simon Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor Books, 1999.
- [10] Gary Stix. Profile: David Huffman. *Scientific American*, pages 54–58, 1991.
- [11] Peter Wayner. *Disappearing Cryptography: Being and Nothingness on the Net*. Academic Press, Inc., 1996.
- [12] Philip Zimmermann. Testimony to the subcommittee on science, technology, and space of the us senate committee on commerce, science, and transportation, 1996.

Appendix A

References

Today, by radio, and also on giant hoardings, a rabbi, an admiral notorious for his links to masonry, a trio of cardinals, a trio, too, of insignificant politicians (bought and paid for by a rich and corrupt Anglo-Canadian banking corporation), inform us all of how our country now risks dying of starvation. A rumor, that's my initial thought as I switch off my radio, a rumor or possibly a hoax. Propaganda, I murmur anxiously - as though, just by saying so, I might allay my doubts - typical politicians' propaganda.

Figure A.1: Sample source text used to generate Huffman coding tree in Figure 3.2

Letter	Percentage	Letter	Percentage
a	8.2	n	6.7
b	1.5	o	7.5
c	2.8	p	1.9
d	4.3	q	0.1
e	12.7	r	6.0
f	2.2	s	6.3
g	2.0	t	9.1
h	6.1	u	2.8
i	7.0	v	1.0
j	0.2	w	2.4
k	0.8	x	0.2
l	4.0	y	2.0
m	2.4	z	0.1

Table A.1: Frequency distribution of English language used to generate Figure 2.3 [9]

Order (n)	Average Execution Time
2	0.0106
3	0.02
4	0.0334
5	0.0808
6	0.0882
7	0.1756
8	0.359

Table A.2: Average timing test results, Lab3.txt

Order (n)	Average Execution Time
2	0.0402
3	0.053
4	0.08
5	0.1318
6	0.21
7	0.3944
8	0.6998

Table A.3: Average timing test results, chapter5.txt

Order (n)	Average Execution Time
2	0.2642
3	0.353
4	0.4724
5	0.6142
6	0.9462
7	1.5064
8	3.7798

Table A.4: Average timing test results, macbeth.txt

Order (n)	Average Execution Time
2	4.289
3	4.6548
4	4.9312
5	4.8844
6	5.3246
7	6.2588
8	8.1654

Table A.5: Average timing test results, inferno.txt

Appendix B

Readability Survey

B.1 Introduction

The following survey was distributed to several participants. The purpose of the survey is to subjectively evaluate the effectiveness of the mimicry algorithm implemented in this thesis.

B.2 Survey Contents

Readability Survey

Dear Participant,

Thank you in advance for your participation in this questionnaire. The purpose of this survey is to subjectively evaluate the "readability" of various texts generated by the mimicry algorithm employed in my undergraduate thesis.

For each of the following sections, you will be asked to read a short sample of text and then evaluate it based upon the following criteria:

Logical Flow:

This section requires that you evaluate whether or not the text reads in a logical manner. Does the text jump back and forth from one idea to another, or is it consistent?

- 1) Logical Consistency
- Extremely inconsistent
- Slightly inconsistent

- Average (compared to most writing)
- Above average in consistency
- Exceptionally consistent

Verbal Flow:

In this section, try not to worry about the overall meaning of the sample of text; simply concentrate on how each phrase follows from the next. Does it read with a smooth flow?

2) Flow of writing

- Extremely jittered
- Occasional jitters
- Average (compared to most writing)
- Flows very well
- Flows extremely well

Punctuation:

Count the number of punctuation errors you find in this text, examining it as closely as you would when reading in your "normal" reading style.

3) Number of Punctuation errors: _____

Overall Grammar:

How many obvious grammatical mistakes are present in this text?

4) Number of grammatical errors: _____

Style:

If you are familiar with the source text (or other works by the author), how closely do you feel the style of the mimicked text resembles that of the author?

5) Resemblance to author's style

- Are you sure the author didn't write this?
- Very closely resembles the author's writing
- It could possibly be written by the author
- This looks nothing like the author's writing!

6) Are you familiar with the topic of the source text or the jargon present within it?

- Yes No

7) If you were not informed beforehand that this text was

computer-generated, would you have suspected that a computer wrote it?

- Yes No

Appendix C

Program Documentation

C.1 `mimic`

Usage: `mimic <order> <source text> <encode | decode> [output file]`

The *mimic* program is the main program developed in this thesis.

Parameters are defined as follows:

Parameter Name	Required/Optional	Description
<i>order</i>	Required	Specifies the mimicry order, n .
<i>source text</i>	Required	Specifies the source text, S .
<i>encode decode</i>	Required	Specifies whether text is being encoded or decoded.
<i>output file</i>	Optional	Logs the output to a specified file.

Table C.1: Command line parameters for `mimic`

Input is always read from the standard input stream, `stdin`. When keyboard based input is being used, input can be terminated with the EOF character (Ctrl+D on most systems). Input can also be redirected from a file using the `<` (redirection) command.

C.2 freqdist

Usage: `freqdist <input file>`

The *freqdist* program is a simple utility that can be used to calculate the frequency distribution of each character in a source text file.

Appendix D

Source Code

D.1 `mimic.c`

```
//  
// Implementation of a text mimicry algorithm to pass covert data  
// Undergraduate thesis  
// Adam Tenenbaum, Engineering Science 0T2  
//  
// mimic.c – Main mimic program  
//  
  
#include <stdio.h>  
#include <stdlib.h> 10  
#include <unistd.h>  
#include <sys/time.h>  
#include <string.h>  
  
#include "mimic.h"  
#include "error.h"  
#include "hash.h"  
#include "huffman.h"  
#include "bool.h" 20  
  
// Global variables  
int order; // Order of mimicry  
struct hashTable* rootTable; // Root hash table  
struct hashNode** tupleList; // List of order – 1 previous nodes  
BOOL progressDisplay = TRUE; // Display progress of encoding?  
BOOL statisticsDisplay = TRUE; // Display statistics after encoding?  
char* pMessage; // Secret message  
int messageSize = BASE_MESSAGE_SIZE; // Size of secret message  
int nMessageChars = 0; // Number of characters in secret message  
int nOutputChars = 0; // Number of characters in output 30  
  
// Main program loop  
int main(int argc, char *argv[])  
{
```

```

MIMIC_MODE mode; // Encode or decode?

printf( "-----\n" );
printf( "  mimic                               \n" );
printf( "  by Adam Tenenbaum                       \n\n" );
printf( "  Applied steganography using text mimicry \n\n" );
printf( "  usage: mimic <order> <source text> <encode | decode> [output file] \n" );
printf( "  e.g.  mimic 3 source.txt encode message.txt \n" );
printf( "-----\n\n" );

if( argc < 4 )
{
    printf("Invalid number of command-line arguments.\n");
    exit( -1 );
}

order = atoi( argv[1] );

if( order < 1 || order > MAX_ORDER ) {
    printf( "Please enter an order value between 1 and %d\n", MAX_ORDER );
    exit( -1 );
}

// Set appropriate mode
if( strstr( argv[3], "enc" ) ) {
    mode = ENCODE;
}
else if( strstr( argv[3], "dec" ) ) {
    mode = DECODE;
}
else {
    printf( "Please specify either encode or decode mode.\n");
    exit( -1 );
}

// Allocate globals
printf("Allocating memory..");
fflush( stdout );
AllocateGlobals();
printf("done!\n");

if( mode == ENCODE ) {
    // Get secret message from stdin

    printf("Input secret message:\n");
    ReadMessage();
    printf("Done reading secret message!\n");

    /*
    // TEST: print message

    // printf("<Message start>:\n%s\n<Message end>\n", pMessage );

```

```

// TEST: show first two bytes
// CharToByte( message[0] );
// CharToByte( message[1] );
*/
}

// Parse file into tables

printf("Building frequency tables from source text...\n");
ParseSource( argv[2] );

// TEST: display hash table

// Traverse( rootTable, 0 );

// TEST: 1000 words of random text

// RandomText( 1000 );

// Build Huffman Trees

printf("Building Huffman trees from frequency tables...\n");
BuildHuffmanTrees();
printf("Done!\n");

if( mode == ENCODE ) {
    printf("Encoding message...\n\n");
    if( argv[4] == NULL )
        printf("<ENCODED MESSAGE BEGINS HERE>\n\n");
    EncodeMessage( argv[4] );
    if( argv[4] == NULL )
        printf("\n<ENCODED MESSAGE END>\n\n");

    printf("Message encoding complete!\n\n");

    if( statisticsDisplay == TRUE )
        DisplayStatistics();
}

if( mode == DECODE ) {
    printf("Decoding message...\n\n");
    if( argv[4] == NULL )
        printf("<DECODED MESSAGE BEGINS HERE>\n\n");
    DecodeMessage( argv[4] );
    if( argv[4] == NULL )
        printf("\n<DECODED MESSAGE END>\n\n");
    printf("Message decoded!\n\n");
}

// Clean up

// FreeGlobals();

```

```

    exit( 0 );
}

void AllocateGlobals() {

    rootTable = hashTableNew( ROOT_TABLE_SIZE );
                                                                    150

    if( !rootTable ) {
        error("AllocateGlobals: rootTable is NULL");
        exit( -1 );
    }

    tupleList = ( struct hashNode** ) malloc( sizeof( struct hashNode* ) * ( order - 1 ) );

    if( !tupleList ) {
        error("AllocateGlobals: tupleList is NULL");
        exit( -1 );
                                                                    160
    }

    InitTupleList();

    pMessage = ( char* ) malloc( sizeof( char ) * messageSize );

    if( pMessage == NULL ) {
        error("AllocateGlobals: message is NULL");
        exit( -1 );
                                                                    170
    }

    // Initialize message string with null terminator
    pMessage[0] = '\0';
}

void FreeGlobals() {
    printf("FreeGlobals(): Cleaning up. .\n");

    if( rootTable )
        hashTableDelete( rootTable );
                                                                    180

    if( tupleList )
        free( tupleList );

    if( pMessage )
        free( pMessage );
}

void ParseSource( char* fileName ) {
    FILE* fp;
                                                                    190
    unsigned long nTotalWords;
    unsigned long nWords = 0;
    int prevPercentageDone = 0;

    fp = fopen( fileName, "r" );

```

```

if( !fp ) {
    error("ParseSource: Could not open file" );
    exit( -1 ) ;
}
                                                                    200

nTotalWords = PrecountWords( fp );
fp = freopen( fileName, "r", fp );

while( !feof( fp ) ) {
    char* word;

    unsigned long percentageDone = ( nWords * 100 ) / nTotalWords ;

    if( percentageDone > prevPercentageDone ) {
        printf("%ld percent complete.\r", percentageDone );
        fflush( stdout );
        prevPercentageDone = percentageDone;
    }
                                                                    210

    word = ParseWord( fp );

    if( !word )
        break;
                                                                    220

    nWords++;
    InsertWord( word );

    free( word );
}

fclose( fp );
printf("\nDone!\n");
}
                                                                    230

char* ParseWord( FILE* fp ) {
    char* pWord;
    char c;
    int wordSize = BASE_WORD_SIZE;

    if( fp == NULL ) {
        error("ParseWord: NULL fp");
        return( NULL );
    }
                                                                    240

    pWord = ( char* ) malloc( sizeof( char ) * wordSize );

    if( !pWord ) {
        error("ParseWord: NULL pWord" );
        return( NULL );
    }

    // Read over whitespace
    while( ( c = fgetc( fp ) ) != EOF ) {
        if( !isWhitespace( c ) ) {
                                                                    250

```

```

    break;
}
}

if( c == EOF )
    return( NULL );

if( isPunctuation( c ) ) {
    pWord[0] = c;
    pWord[1] = '\0';
}
// Otherwise, character string
else {
    int wordIndex = 0;

    while( c != EOF ) {
        if( isWhitespace( c ) || isPunctuation( c ) ) {
            // Put it back
            ungetc( c, fp );
            break;
        }
        else {
            // Make sure we don't overrun the buffer -- and subtract - 1 for
            // the null terminator
            if( wordIndex >= wordSize - 1 ) {
                wordSize += BASE_WORD_SIZE;

                pWord = ( char* ) realloc( pWord, sizeof( char ) * wordSize );

                if( !pWord ) {
                    error("ParseWord: realloc failed");
                    exit( -1 );
                }
            }
            pWord[wordIndex++] = c;
        }
        c = fgetc( fp );
    }
    pWord[wordIndex] = '\0';
}

return( pWord );
}

BOOL isWhitespace( char c ) {
    switch( c ) {
        case ' ':
        case '\t':
        case '\n':
        case '\r':
            return( TRUE );
        default:
            return( FALSE );
    }
}

```

260

270

280

290

300

```

}

BOOL isPunctuation( char c ) {
    switch( c ) {
        case ',':
        case ':':
        case ';':
        case '!':
        case '?':
        case '\':
        case '(':
        case ')':
        case '[':
        case ']':
        case '{':
        case '}':
            return( TRUE );
        default:
            return( FALSE );
    }
}

void InsertWord( char* word ) {
    struct hashNode* rootEntry;
    int i;

    rootEntry = hashTableInsert( rootTable, word );

    if( !rootEntry ) {
        error("InsertWord: rootEntry is NULL");
        exit( -1 );
    }

    if( rootEntry->frequency == 0 ) {
        rootEntry->frequency = 1;
        rootEntry->pWord = (char *)strdup( word );
    }

    for( i = 0; i < order - 1; i++ ) {
        struct hashNode* pTuple = tupleList[i];
        struct hashNode* tupleEntry;

        if( !pTuple )
            continue;

        if( !pTuple->pNextWordTable ) {
            int newTableSize = ( SUB_TABLE_BASE_SIZE << i );

            //    printf("Creating table of size %d\n", newTableSize );

            pTuple->pNextWordTable = hashTableNew( newTableSize );

            if( !pTuple->pNextWordTable ) {

```

```

        error("InsertWord: pTuple->pNextWordTable is NULL");
        exit( -1 );
    }
}
360

tupleEntry = hashTableInsert( pTuple->pNextWordTable, word );

if( !tupleEntry ) {
    error("InsertWord: tupleEntry is NULL");
    exit( -1 );
}
370

if( tupleEntry->frequency == 0 ) {
    tupleEntry->frequency = 1;
    tupleEntry->pHashNode = rootEntry;
}

tupleList[i] = tupleEntry;
}

for( i = 0; i < order - 2; i++ ) {
    tupleList[i] = tupleList[i+1];
}
380

tupleList[order - 2] = rootEntry;
}

unsigned long PrecountWords( FILE* fp ) {
    unsigned long nTotalWords;

    nTotalWords = 0;
    390

    while( !feof( fp ) ) {
        char* pWord;

        pWord = ParseWord( fp );

        if( pWord != NULL )
            nTotalWords++;
    }
    printf("Total number of words: %ld\n", nTotalWords );
    return( nTotalWords );
    400
}

void Traverse( struct hashTable* h, int traversalDepth ) {
    int i, depth;
    struct hashNode* pNode;

    if( !h )
        return;

    for( i = 0; i < h->nSlots; i++ ) {
        for( pNode = h->pTable[i]; pNode != NULL; pNode = pNode->pNextEntry ) {
            410

```

```

    for( depth = 0; depth < traversalDepth; depth++ ) {
        printf("-");
    }

    printf("%s (%d)", NodeToString( pNode ), pNode->frequency );

    printf("\n");
    Traverse( pNode->pNextWordTable, traversalDepth + 1 );
}
}
}

void RandomText( int nWords ) {
    struct hashNode* pNode = NULL;
    struct hashNode* pNextNode = NULL;

    pNode = RandomNode( rootTable );

    printf("%s ", NodeToString( pNode ) );
    fflush( stdout );

    InitTupleList();

    tupleList[order - 2] = pNode;

    while( --nWords > 0 ) {
        pNode = NextInTupleList();

        if( !pNode->pNextWordTable ) {
            printf("no nextwordtable for %s\n", NodeToString( pNode ) );
            exit( -1 );
        }

        // Pick a next word at random
        pNextNode = RandomNode( pNode->pNextWordTable );

        printf("%s ", NodeToString( pNextNode ) );
        fflush( stdout );

        UpdateTupleList( pNextNode->pHashNode );
    }
}

struct hashNode* RandomNode( struct hashTable* t ) {
    struct timeval time;    // Time of day
    int slot;
    struct hashNode* pNode = NULL;

    /* Get time of day to seed randomizer */
    gettimeofday( &time, 0 );
    /* Seed randomizer */
    srand( time.tv_sec );

    // Get the first word

```

420

430

440

450

460

```

do {
    slot = ( rand() % t->nSlots );
    pNode = t->pTable[slot];
} while( !pNode );
                                        470

return( pNode );
}

struct hashNode* RandomStartNode() {
    struct hashNode* pNode;

    pNode = RandomNode( rootTable );

    while( pNode->pWord[0] < 'A' || pNode->pWord[0] > 'Z' )
        pNode = RandomNode( rootTable );
                                        480

    return( pNode );
}

void BuildHuffmanTrees() {
    int i;
    struct hashNode* pNode;

    for( i = 0; i < rootTable->nSlots; i++ ) {
        for( pNode = rootTable->pTable[i]; pNode != NULL; pNode = pNode->pNextEntry ) {
            if( pNode->pNextWordTable )
                pNode->pHuffmanTreeRoot = CreateHuffmanTree( pNode->pNextWordTable );
        }
    }
}
                                        490

void ReadMessage() {
    char s[MAX_STRING_LENGTH];
                                        500

    while( fgets( s, MAX_STRING_LENGTH, stdin ) ) {
        nMessageChars += strlen( s );
        if( nMessageChars >= messageSize ) {
            messageSize *= 2;

            pMessage = ( char* ) realloc( pMessage, sizeof( char ) * messageSize );

            if( pMessage == NULL ) {
                error("ReadMessage: realloc failed");
                exit( -1 );
            }
        }
        // Add the new string to message
        strcat( pMessage, s );
    }
}

void CharToByte( char c ) {
    int i;
                                        520

```

```

for( i = 0; i < sizeof( char ) * 8; i++ ) {
    if( ( c >> i ) & 1 )
        printf("1");
    else
        printf("0");
}
printf("\n");
}

```

530

```

void EncodeMessage( char* fileName ) {
    struct hashNode* pNode;
    struct huffmanTreeNode* pTreeNode;
    FILE* fp;
    int i;
    char* pMessagePtr = pMessage;
    char* pPrevWord = NULL;

    if( fileName ) {
        fp = fopen( fileName, "w" );

```

540

```

        if( !fp ) {
            error("EncodeMessage: Could not open file" );
            exit( -1 ) ;
        }
    }
    // No filename specified -- output to stdout
    else {
        fp = stdout;
    }

```

550

```

    // Get a random starting point
    pNode = RandomNode( rootTable );

    // Display the "seed" word
    PrintFormatted( pNode->pWord, NULL, fp );

    // Update "previous word"
    pPrevWord = pNode->pWord;

```

560

```

    pTreeNode = pNode->pHuffmanTreeRoot;

    // Reinitialize tupleList
    InitTupleList();
    tupleList[order - 2] = pNode;

    // Loop through message string
    while( *pMessagePtr != '\0' ) {
        char c = *pMessagePtr;

```

570

```

        // Loop through single character
        for( i = 0; i < sizeof( char ) * 8; i++ ) {
            // Check if pTreeNode has a pHashNode -- if so, it is a leaf node
            while( pTreeNode->pHashNode ) { // Leaf node -- need to print and update

```

```

char* pWord; // Pointer to the word being displayed

// Get the string for the word being displayed
pWord = NodeToString( pTreeNode->pHashNode );

PrintFormatted( pWord, pPrevWord, fp ); // 580

// Update "previous word"
pPrevWord = pWord;

// if( pTreeNode->pHashNode->pHuffmanCodeword )
// printf("%s: %s\n", pWord, pTreeNode->pHashNode->pHuffmanCodeword );

// Update the tuple list, putting the current node at the end
UpdateTupleList( pTreeNode->pHashNode->pHashNode ); // 590

// Update node to get next in tuple list
pNode = NextInTupleList();
pTreeNode = pNode->pHuffmanTreeRoot;

// If we run out of stuff to encode, start again
// at another random location
// We'll have to check this later when we decode

if( pTreeNode == NULL ) { // 600
    // Get a random starting point
    pNode = RandomNode( rootTable );
    pTreeNode = pNode->pHuffmanTreeRoot;

    // Display the "seed" word
    PrintFormatted( pNode->pWord, pPrevWord, fp );

    // Update "previous word"
    pPrevWord = pNode->pWord;

    // Reinitialize tupleList // 610
    InitTupleList();
    tupleList[order - 2] = pNode;
}
}

// Now, pTreeNode points to a node that does not have a word
// associated with it -- therefore, it contains pointers to child
// nodes and a bit can be encoded by traversing the huffman tree

// Define a '1' to be a move to the left child and a '0' to be // 620
// a move to the right child
if( ( c >> i ) & 1 ) {
    pTreeNode = pTreeNode->pLeftChild;
}
// '0' - right child
else {
    pTreeNode = pTreeNode->pRightChild;
}
}

```

```

    }
    pMessagePtr++;
}
}
630

void PrintFormatted( char* pWord, char* pPrevWord, FILE* fp ) {
    int numSpaces; // Number of spaces
    int spaceCounter; // Counter of how many spaces have been displayed

    numSpaces = LeadingSpaces( pWord, pPrevWord );
    // Print an appropriate number of spaces
    for( spaceCounter = 0; spaceCounter < numSpaces; spaceCounter++ )
        fprintf( fp, " ");
    // Display the word
    fprintf( fp, "%s", pWord );
    // Make sure fp is flushed
    fflush( fp );
    // Add number of characters to nOutputChars
    nOutputChars += strlen( pWord ) + numSpaces;
}
640

struct hashNode* NextInTupleList() {
    int i;

    for( i = 0; i < order - 1; i++ ) {
        if( tupleList[i] ) {
            return( tupleList[i] );
        }
    }
    return( NULL );
}
650

void InitTupleList() {
    int i;

    for( i = 0; i < order - 1; i++ )
        tupleList[i] = NULL;
}
660

void UpdateTupleList( struct hashNode* pNode ) {
    int i;

    for( i = 0; i < order - 2; i++ ) {
        if( tupleList[i+1] ) {
            tupleList[i] = hashTableLookup( tupleList[i+1]->pNextWordTable, pNode->pWord );
        }
    }
    tupleList[order - 2] = pNode;
}
670
680

```

```

}

char* NodeToString( struct hashNode* pNode ) {
    if( pNode->pWord )
        return( pNode->pWord );
    else
        return( pNode->pHashNode->pWord );
}
690

void DecodeMessage( char* fileName ) {
    FILE* fp;
    char messageChar = 0;
    int bitShift = 0;
    char* pWord;
    struct hashNode* pNode;

    if( fileName ) {
        fp = fopen( fileName, "w" );
600

        if( !fp ) {
            error("DecodeMessage: Could not open file" );
            exit( -1 ) ;
        }
    }
    // No filename specified -- output to stdout
    else {
        fp = stdout;
610
    }

    // Get the starting point

    // Read in a word from standard input
    pWord = ParseWord( stdin );

    if( pWord == NULL ) {
        error("DecodeMessage: no message entered!");
        return;
620
    }

    pNode = hashTableLookup( rootTable, pWord );

    if( pNode == NULL ) {
        error("DecodeMessage: decoding failed!");
        return;
    }

    // Free pWord -- done using it
630
    free( pWord );

    // Reinitialize tupleList
    InitTupleList();
    tupleList[order - 2] = pNode;
}

```

```

// Loop through message string as long as there are words
while( ( pWord = ParseWord( stdin ) ) != NULL ) {

    // Update node to get next in tuple list
    pNode = NextInTupleList();

    // Most likely scenario is that pNode has a valid next word table
    if( pNode->pNextWordTable != NULL ) {
        pNode = hashTableLookup( pNode->pNextWordTable, pWord );
    }
    // If not, we start at the root again
    else {
        pNode = hashTableLookup( rootTable, pWord );

        // Reinitialize tupleList
        InitTupleList();
        tupleList[order - 2] = pNode;
        continue;
    }

    if( pNode->pHuffmanCodeword != NULL ) {
        char* pCodeword = pNode->pHuffmanCodeword;

//    printf("%s: %s\n", NodeToString( pNode ), pCodeword );
//    fflush( stdout );

        while( *pCodeword != '\0' ) {
            if( *pCodeword++ == '1' )
                messageChar |= 1 << bitShift;

            bitShift++;

            // Done with the character
            if( bitShift >= ( sizeof( char ) * 8 ) ) {
                fputc( messageChar, fp );
                fflush( fp );
                messageChar = 0;
                bitShift = 0;
            }
        }
    }

    // Update the tuple list, putting the current node at the end
    UpdateTupleList( pNode->pHashNode );

    // Free the word allocated by ParseWord
    free( pWord );
}

// The next two functions deal with leading and trailing spaces
// This is necessary to make sure spacing is right in terms of
// punctuation.
// Normal words will have one space before them; punctuation will

```

```

// have none.
// Normal words will not have space after them; periods will have one
// space.

int LeadingSpaces( char* pWord, char* pPrevWord ) {
    char lastPrevChar;
    char firstChar;

    if( pPrevWord == NULL )
        return( 0 );
        800

    // Last character in previous word is at index length(pPrevWord) - 1
    lastPrevChar = pPrevWord[strlen( pPrevWord ) - 1];

    firstChar = pWord[0];

    switch( lastPrevChar ) {
        case '.' :
            switch( firstChar ) {
                case '.' :
                    return( 0 );
                    810
                default:
                    return( 2 );
            }
        case '?' :
        case '!' :
            return( 2 );

        case ',' :
            switch( firstChar ) {
                case '\"' :
                    return( 0 );
                    820
                default:
                    return( 1 );
            }

        case '\"' :
            return( 1 );

        case '(' :
        case '[' :
        case '{' :
            return( 0 );
            830

        case ')' :
        case ']' :
        case '}' :
        case ';' :
        case ':' :
            840
        default:
            if( isPunctuation( firstChar ) )
                return( 0 );
            else
                return( 1 );
    }
}

```

```

}
}

// Display coding statistics
void DisplayStatistics() {
    printf("Message: %d bits, Output: %d bits\n", nMessageChars * 8, nOutputChars * 8 );      850
    printf("Expansion Factor: %.2f\n", ( double )( nOutputChars ) / ( double ) nMessageChars );
}

```

D.2 mimic.h

```

//
// Implementation of a text mimicry algorithm to pass covert data
// Undergraduate thesis
// Adam Tenenbaum, Engineering Science OT2
//
// mimic.h - Main mimic program
//

#ifdef `MIMIC`H`
#define `MIMIC`H`                                     10

// Maximum string length
#define MAX`STRING`LENGTH 255

// Maximum allowable order
#define MAX`ORDER` 8

// Size of root hash table
#define ROOT`TABLE`SIZE ( 1 << 10 )                                     20

// Start size of smallest table
#define SUB`TABLE`BASE`SIZE ( 1 << 2 )

// Start size of message
#define BASE`MESSAGE`SIZE 256

// Start size of word
#define BASE`WORD`SIZE 20

// Enumeration of possible mimic modes                                     30
typedef enum { ENCODE = 0, DECODE = 1 } MIMIC`MODE`;

// Move this down here to avoid problems
#include "hash.h"

// Allocate global variables
void AllocateGlobals();

// Parse file into tables
void ParseSource( char* fileName );                                     40

```

```

// Clean up global variables
void FreeGlobals();

// Read the secret message from stdin
void ReadMessage();

// Insert a word into the root table
void InsertWord( char* word );

// Parse a single word from fp
char* ParseWord( FILE* fp );

// Is c whitespace?
BOOL isWhitespace( char c );

// Is c punctuation?
BOOL isPunctuation( char c );

// Count the words in file fp
unsigned long PrecountWords( FILE* fp );

// Traverse all the nodes in hash table
void Traverse( struct hashTable* pHashTable, int traversalDepth );

// Print out n words of random text
void RandomText( int nWords );

// Get a random node in pHashTable
struct hashNode* RandomNode( struct hashTable* pHashTable );

// Get a random start node (beginning with a capital)
struct hashNode* RandomStartNode();

// Build the huffman trees for all nodes in the root table
void BuildHuffmanTrees();

// Print out c as a byte
void CharToByte( char c );

// Encode the secret message, outputting to fileName
void EncodeMessage( char* fileName );

// Print a word in "formatted form" – with leading and trailing spaces
void PrintFormatted( char* pWord, char* pPrevWord, FILE* fp );

// Get the first entry in tupleList
struct hashNode* NextInTupleList();

// Clear the contents of tupleList
void InitTupleList();

// Update the tuple list, inserting pNode at the bottom
void UpdateTupleList( struct hashNode* pNode );

```

```

// Print out the string representation of pNode
char* NodeToString( struct hashNode* pNode );

// Decode the secret message, outputting to fileName
void DecodeMessage( char* fileName );

// Return the number of leading spaces before pWord (based on current and previous word)
int LeadingSpaces( char* pWord, char* pPrevWord );

// Display coding statistics
void DisplayStatistics();
#endif

```

D.3 hash.c

```

//
// Implementation of a text mimicry algorithm to pass covert data
// Undergraduate thesis
// Adam Tenenbaum, Engineering Science OT2
//
// hash.c – hash related data structures + functions
//

#include <stdlib.h>
#include <string.h>

#include "bool.h"
#include "hash.h"
#include "error.h"

// Hashing function
// Returns a hash key for word
//
// Let's try.. ELFHash -- apparently this is a good algorithm
// to generate a uniform hash for character strings

unsigned long hash( char* pWord ) {
    unsigned long g;
    unsigned long h = 0; // Hash index

    // Loop through characters in string
    while (*pWord) {
        h = (h << 4) + *pWord++;
        if( ( g = h & 0xf0000000 ) )
            h ^= g >> 24;
        h &= ~g;
    }
    return( h );
}

// Allocate memory for a new hash table
struct hashTable* hashTableNew( int nSlots ) {

```

```

struct hashTable* pNewTable; // Pointer to new table

pNewTable = ( struct hashTable* ) malloc( sizeof( struct hashTable ) );
                                                                    40

if( pNewTable == NULL ) {
    error("hashTableNew: Could not allocate memory for hash table (struct hashTable)");
    return( NULL );
}

pNewTable->nSlots = nSlots;
pNewTable->pTable = ( struct hashNode** ) malloc( sizeof( struct hashNode* ) * nSlots );

if( pNewTable->pTable == NULL ) {
    error("hashTableNew: Could not allocate memory for hash table (struct hashNode**)");
    free( pNewTable );
    return( NULL );
}

// Initialize the new hash table
hashTableInit( pNewTable );
return( pNewTable );
}
                                                                    60

// Deallocate memory for a hash table
void hashTableDelete( struct hashTable* pHashTable ) {
    struct hashNode** pTable; // Pointer to array in hash table

    // If pHashTable points to nothing, we're done
    if( pHashTable == NULL )
        return;

    pTable = pHashTable->pTable;
                                                                    70

    // If pTable exists, delete its entries
    if( pTable ) {
        int i; // Index into pTable

        for( i = 0; i < pHashTable->nSlots; i++ ) {
            struct hashNode* pDeleteNode; // Node to be deleted
            struct hashNode* pNode; // Current node

            // Current node is first node at table index i
            pNode = pTable[i];
                                                                    80

            // Go through all nodes at index i
            while( pNode ) {
                pDeleteNode = pNode;
                pNode = pNode->pNextEntry;

                // Delete node
                hashNodeDelete( pDeleteNode );
            }
        }
    }
}
                                                                    90

```

```

    // Free array
    free( pTable );
}

// Free hash table structure
free( pHashTable );
return;
}

```

100

```

// Initialize the hash table
void hashTableInit( struct hashTable* pHashTable ) {
    int i; // Index into pTable
    struct hashNode** pTable; // Table (array) of hash entries

    // If pHashTable is null we're done
    if( pHashTable == NULL )
        return;

    pTable = pHashTable->pTable;

    // If no pTable, nothing to do
    if( pTable == NULL )
        return;

    // Loop through entries in hash table, initializing pointers to null
    for( i = 0; i < pHashTable->nSlots; i++ ) {
        pTable[i] = NULL;
    }
}

```

110

120

```

// Insert an item into the hash table
struct hashNode* hashTableInsert( struct hashTable* pHashTable, char* pWord ) {
    int slot; // Return slot from hash function
    int index; // Slot in hash table
    struct hashNode* pExistingNode; // Existing node in hash table
    struct hashNode* pNewNode; // New node
    struct hashNode** pTable; // Table (array) of hash entries

    if( pHashTable == NULL ) {
        error("hashTableInsert: hashTable is NULL");
        return( NULL );
    }
    pTable = pHashTable->pTable;

    if( pTable == NULL ) {
        error("hashTableInsert: hashTable->pTable is NULL");
        return( NULL );
    }

    if( ( slot = hash( pWord ) ) < 0 ) {
        error("hashTableInsert: hash index < 0");
        return( NULL );
    }
}

```

130

140

```

}

// Get the array index of the hash value by taking the modulo of the
// hash value with the number of slots in the table
index = ( slot % pHashTable->nSlots );
150

pExistingNode = pTable[ index ];

// If there's already an entry in that slot, check if
// the word is already in the table.
while( pExistingNode ) {
    // Is the existing node we're looking at the word we're inserting?
    // If so, simply increment the count of this word and return it
    if( nodeIsWord( pExistingNode, pWord ) ) {
        pExistingNode->frequency++;
        return( pExistingNode );
    }
    // Otherwise, look at the next node
    pExistingNode = pExistingNode->pNextEntry;
}

// Through the loop - didn't find the word.
// existingNode = NULL

// Allocate for the new node
pNewNode = hashNodeNew();
170

if( pNewNode == NULL ) {
    error("hashTableInsert: newNode is NULL!");
    return( NULL );
}

// If there was a collision, place the new node at the head of the list
// TODO: Change this to sort by frequency
if( pTable[ index ] )
    pNewNode->pNextEntry = pTable[ index ];
180

// Add the new node to the hash table
pTable[ index ] = pNewNode;

return( pNewNode );
}

// Access a word in the hash table
struct hashNode* hashTableLookup( struct hashTable* pHashTable, char* pWord ) {
190
    int slot;          // Return slot from hash function
    int index;        // Slot in hash table
    struct hashNode* pEntry; // Entry in hash table
    struct hashNode** pTable; // Table (array) of hash entries

    if( pHashTable == NULL ) {
        error("hashTableLookup: hashTable is NULL");
        return( NULL );
    }
}

```

```

pTable = pHashTable->pTable;
200

if( pTable == NULL ) {
    error("hashTableLookup: hashTable->pTable is NULL");
    return( NULL );
}

if( ( slot = hash( pWord ) ) < 0 ) {
    error("hashTableLookup: hash index < 0");
    return( NULL );
}
210

index = ( slot % pHashTable->nSlots );

pEntry = pTable[ index ];

// If there's already an entry in that slot, check if
// the word is already in the table.
while( pEntry ) {
    // Check if word is in the table; if so, return it
    if( nodeIsWord( pEntry, pWord ) ) {
        return( pEntry );
    }
    // Otherwise, check the next word
    pEntry = pEntry->pNextEntry;
}

// If the word wasn't found, we'll get here.. return NULL
return( NULL );
230
}

// Allocate memory for a new hash table entry
struct hashNode* hashNodeNew() {
    struct hashNode* pNewNode; // New node being created

    // Create a new node
    pNewNode = ( struct hashNode* ) malloc( sizeof( struct hashNode ) );

    // Make sure memory was malloced
    if( pNewNode == NULL ) {
        error("hashNodeNew: Could not allocate memory for hashNode");
        return( NULL );
    }
    // Initialize member variables
    pNewNode->pNextEntry = NULL;
    pNewNode->frequency = 0;

    // Deal with these outside this function
    pNewNode->pWord = NULL;
    pNewNode->pHashNode = NULL;
    pNewNode->pNextWordTable = NULL;
250
}

```

```

pNewNode->pHuffmanTreeRoot = NULL;
pNewNode->pHuffmanCodeword = NULL;

return( pNewNode );
}

// Deallocate memory for a hash entry
void hashNodeDelete( struct hashNode* pNode ) {
    // Nothing to do if null pointer
    if( pNode == NULL )
        return;

    // Free word
    if( pNode->pWord )
        free( pNode->pWord );

    // Free hash table
    if( pNode->pNextWordTable )
        hashTableDelete( pNode->pNextWordTable );

    // Free Huffman Codeword
    if( pNode->pHuffmanCodeword )
        free( pNode->pHuffmanCodeword );

    // Free the node itself
    free( pNode );
}

// Return TRUE if pNode is a reference to "pWord"
// FALSE otherwise
BOOL nodeIsWord( struct hashNode* pNode, char* pWord ) {
    if( pNode == NULL ) {
        error("nodeIsWord: null pNode");
        return( FALSE );
    }

    if( pWord == NULL ) {
        error("nodeIsWord: null pWord");
        return( FALSE );
    }

    // If word actually contains the string
    if( pNode->pWord != NULL ) {
        // Check if the words match up
        if( !strcmp( pNode->pWord, pWord ) )
            return( TRUE );
        else
            return( FALSE );
    }
    else {
        struct hashNode* pHashNode; // "Root entry" hash node pointed to by

```

```

        // this node

// Make sure this node points to a valid node
if( ( pHashNode = pNode->pHashNode ) == NULL ) {
    error("nodeIsWord: null pHashNode");
    return( FALSE );
}

// Make sure the root entry has a word in it
if( pHashNode->pWord == NULL ) {
    error("nodeIsWord: pHashNode has no pWord!");
    return( FALSE );
}

// Check if the words match up
if( !strcmp( pHashNode->pWord, pWord ) )
    return( TRUE );
else
    return( FALSE );
}
}

```

310

320

D.4 hash.h

```

//
// Implementation of a text mimicry algorithm to pass covert data
// Undergraduate thesis
// Adam Tenenbaum, Engineering Science OT2
//
// hash.h - Structures and functions involved in the manipulation and
//          storage of hash tables
//
#ifndef __HASH_H__
#define __HASH_H__

#include "bool.h"

// Hash table structure
struct hashTable
{
    // Number of slots in hash table
    int nSlots;

    // Array of hash node pointers - entries in hash table
    struct hashNode** pTable;
};

// Single node in the hash table
struct hashNode
{

```

10

20

```

// Pointer to the next node (used if collision on a slot)
// (open hashing)
struct hashNode* pNextEntry; 30

// In any given hashNode, one of the two following pointers
// will be NULL; the other will be valid.
// If pWord is null, pHashnode points to a node with a valid pWord

// Pointer to word
char* pWord;

// Pointer to hash node
struct hashNode* pHashNode; 40

// Frequency of word's appearance in text
int frequency;

// Table of words that can follow this node
struct hashTable* pNextWordTable;

// Root of huffman tree that contains possible next words
struct huffmanTreeNode* pHuffmanTreeRoot; 50

// If this node is a leaf node in a huffman tree (which all but
// the root nodes are), this stores the codeword that is encoded
// by the word
char* pHuffmanCodeword;
};

// It's strange to have this here, but it fixes some recursive dependency
// compile errors
#include "huffman.h" 60

// Allocate memory for a new hash table
struct hashTable* hashTableNew( int nSlots );

// Deallocate hash table memory
void hashTableDelete( struct hashTable* pHashTable );

// Hashing function
unsigned long hash( char* pWord );

// Initialize the hash table 70
void hashTableInit( struct hashTable* pHashTable );

// Insert an item into the hash table
struct hashNode* hashTableInsert( struct hashTable* pHashTable, char* pWord );

// Access a word in the hash table
struct hashNode* hashTableLookup( struct hashTable* pHashTable, char* pWord );

// Allocate memory for a new hash node 80
struct hashNode* hashNodeNew();

```

```

// Deallocate memory for a hash node
void hashNodeDelete( struct hashNode* pNode );

// Return TRUE if pNode either contains pWord, or has a reference
// to a node that contains pWord
BOOL nodeIsWord( struct hashNode* pNode, char* pWord );

#endif

```

D.5 huffman.c

```

//
// Implementation of a text mimicry algorithm to pass covert data
// Undergraduate thesis
// Adam Tenenbaum, Engineering Science OT2
//
// huffman.c – Huffman tree management
//

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include "huffman.h"
#include "error.h"
#include "hash.h"
#include "bool.h"

// Create a huffman Tree for a given hash table
struct huffmanTreeNode* CreateHuffmanTree( struct hashTable* pHashTable ) {
    int i;
    struct hashNode* pHashNode;
    struct huffmanTreeNode* pTreeNode;
    struct huffmanListNode* pListNode;
    struct huffmanListNode* pListHead = NULL; // Head of list

    // Go through all the entries in the table
    for( i = 0; i < pHashTable->nSlots; i++ ) {
        for( pHashNode = pHashTable->pTable[i]; pHashNode != NULL;
            pHashNode = pHashNode->pNextEntry ) {

            // First, generate the Huffman trees for the nodes under this one
            if( pHashNode->pNextWordTable )
                pHashNode->pHuffmanTreeRoot = CreateHuffmanTree( pHashNode->pNextWordTable );

            // Now, generate a huffmanTreeNode out of this node
            pTreeNode = huffmanTreeNodeNew();
            pTreeNode->pHashNode = pHashNode;
            pTreeNode->frequency = pHashNode->frequency;

            // And make a huffmanListNode too

```

```

pListNode = huffmanListNodeNew( pTreeNode );

// If the list of huffmanListNodes has a head, insert by increasing frequency
if( pListHead )
    pListHead = huffmanListInsert( pListNode, pListHead );

// If the list doesn't have a head, set this to be the head
else
    pListHead = pListNode;
}
}

// Now, pListHead contains the head of the list of huffmanTreeNode leaf nodes
// Build a Huffman Tree from these nodes

// Loop for as long as there are at least two nodes in the list
while( pListHead->pNext ) {
    struct huffmanListNode* pLeft; // "Left" child in Huffman tree
    struct huffmanListNode* pRight; // "Right" child in Huffman tree

    // Take the two first entries in the list (by definition, the two
    // lowest frequency ones) and combine them to make a new node

    pLeft = pListHead;
    pRight = pListHead->pNext;

    // Now, generate a huffmanTreeNode with the two child nodes pointed
    // to by pLeft and pRight

    pTreeNode = huffmanTreeNodeNew();
    pTreeNode->pLeftChild = pLeft->pTreeNode;
    pTreeNode->pRightChild = pRight->pTreeNode;

    // Frequency of new node = frequency of left + right
    pTreeNode->frequency = pTreeNode->pLeftChild->frequency +
        pTreeNode->pRightChild->frequency;

    // Update list head
    pListHead = pRight->pNext;

    // Get rid of those two nodes
    huffmanListNodeDelete( pLeft );
    huffmanListNodeDelete( pRight );

    // Make a huffmanListNode from the new treeNode
    pListNode = huffmanListNodeNew( pTreeNode );

    // If the list of huffmanListNodes has a head, insert by increasing frequency
    if( pListHead )
        pListHead = huffmanListInsert( pListNode, pListHead );

    // If the list doesn't have a head, set this to be the head
    else
        pListHead = pListNode;
}
}

```

```

}

// Traverse the tree, generating the codewords
GenerateCodewords( pListHead->pTreeNode, NULL );
                                                                    100

// Return the root of the newly created Huffman tree
return( pListHead->pTreeNode );
}

// Traverse the tree, generating the codewords
void GenerateCodewords( struct huffmanTreeNode* pTreeNode, char* pCodeword ) {

// If this is a leaf node, set its codeword to pCodeword
if( pTreeNode->pHashNode ) {
    if( pCodeword ) {
        pTreeNode->pHashNode->pHuffmanCodeword = ( char* ) strdup( pCodeword );
    }
}
                                                                    110
// Otherwise, it has children -- call GenerateCodewords recursively
else {
    char* pNewCodeword; // New codeword
    int codewordLength; // Length of existing codeword

// If the parent has a codeword, use it as the base of the new one
if( pCodeword != NULL ) {
    codewordLength = strlen( pCodeword );
    pNewCodeword = ( char* ) malloc( sizeof( char ) * ( codewordLength + 2 ) );
                                                                    120

    strcpy( pNewCodeword, pCodeword );
    pNewCodeword[codewordLength + 1] = '\0';
}
// Otherwise, start with an empty codeword
else {
    pNewCodeword = ( char* ) malloc( sizeof( char ) * 2 );
    pNewCodeword[1] = '\0';
    codewordLength = 0;
                                                                    130
}

// Add a "1" for the left traversal
pNewCodeword[codewordLength] = '1';
GenerateCodewords( pTreeNode->pLeftChild, pNewCodeword );

// Add a "0" for the right traversal
pNewCodeword[codewordLength] = '0';
GenerateCodewords( pTreeNode->pRightChild, pNewCodeword );
                                                                    140

// We're done with the codeword; free it
free( pNewCodeword );
}
}

// Insert into a list by increasing frequency
// Return the head of the list
struct huffmanListNode* huffmanListInsert( struct huffmanListNode* pInsertNode,

```

```

                                struct huffmanListNode* pHead ) {
struct huffmanListNode* pNode;      // Current node
struct huffmanListNode* pTrailingNode = NULL; // Trailing node

// Check that the list has a head
if( pHead == NULL ) {
    error("huffmanListInsert: NULL pHead");
    return( NULL );
}

// Loop through nodes, breaking when one is found with a higher frequency
for( pNode = pHead; pNode != NULL; pNode = pNode->pNext ) {
    if( pNode->frequency > pInsertNode->frequency )
        break;

    // Update trailing node
    pTrailingNode = pNode;
}

// Insert anywhere in list except head
if( pTrailingNode ) {
    pTrailingNode->pNext = pInsertNode;
    pInsertNode->pNext = pNode;
    return( pHead );
}

// Insert at head (no trailing node)
else {
    pInsertNode->pNext = pNode;
    return( pInsertNode );
}
}

// Allocate memory for a new Huffman Tree Node
struct huffmanTreeNode* huffmanTreeNodeNew()
{
    struct huffmanTreeNode* pNewNode; // Pointer to new node

    // Allocate memory
    pNewNode = ( struct huffmanTreeNode* ) malloc( sizeof( struct huffmanTreeNode ) );

    // Make sure memory alloc went as planned
    if( pNewNode == NULL ) {
        error("huffmanTreeNodeNew: Could not allocate memory for huffmanTreeNode");
        return( NULL );
    }

    // Initialize member variables
    pNewNode->pHashNode = NULL;
    pNewNode->pLeftChild = NULL;
    pNewNode->pRightChild = NULL;
    pNewNode->frequency = 0;
}

```

```

    return( pNewNode );
}

// Allocate memory for a new list node
struct huffmanListNode* huffmanListNodeNew( struct huffmanTreeNode* pTreeNode ) {
    struct huffmanListNode* pNewNode; // Pointer to new node
                                        210

    // Make sure treenode is valid
    if( pTreeNode == NULL ) {
        error("huffmanListNodeNew: Received NULL huffmanTreeNode");
        return( NULL );
    }

    // Allocate memory
    pNewNode = ( struct huffmanListNode* ) malloc( sizeof( struct huffmanListNode ) );

    // Make sure memory allocation succeeded
    if( pNewNode == NULL ) {
        error("huffmanListNodeNew: Could not allocate memory for huffmanListNode");
        return( NULL );
    }
                                        220

    // Initialize member variables
    pNewNode->pNext = NULL;
    pNewNode->pTreeNode = pTreeNode;
    pNewNode->frequency = pTreeNode->frequency;
                                        230

    return( pNewNode );
}

// Deallocate memory for a tree node
void huffmanTreeNodeDelete( struct huffmanTreeNode* pTreeNode ) {
    if( pTreeNode == NULL )
        return;

    free( pTreeNode );
}
                                        240

// Deallocate memory for a tree node
void huffmanListNodeDelete( struct huffmanListNode* pListNode ) {
    if( pListNode == NULL )
        return;

    free( pListNode );
}
                                        250

```

D.6 huffman.h

```

//
// Implementation of a text mimicry algorithm to pass covert data

```

```

// Undergraduate thesis
// Adam Tenenbaum, Engineering Science 0T2
//
// huffman.h - Structures and functions used to manipulate lists and
//             trees containing nodes with frequency information
//
#ifndef `HUFFMAN`H`
10
#define `HUFFMAN`H`

#include "hash.h"

// Huffman Tree node structure
struct huffmanTreeNode {
    // Left child in Huffman tree
    struct huffmanTreeNode* pLeftChild;

    // Right child in Huffman tree
    struct huffmanTreeNode* pRightChild;
20

    // Frequency of node appearance
    int frequency;

    // Pointer to corresponding "root" node
    struct hashNode* pHashNode;
};

// List of Huffman tree nodes
30
struct huffmanListNode {
    // Next in list
    struct huffmanListNode* pNext;

    // Pointer to corresponding tree node
    struct huffmanTreeNode* pTreeNode;

    // Frequency of node appearance
    int frequency;
40
};

// Generate a Huffman tree for the hash table pHashTable
struct huffmanTreeNode* CreateHuffmanTree( struct hashTable* pHashTable );

// Generate the codewords for the leaf nodes in the tree whose root is pTreeNode
void GenerateCodewords( struct huffmanTreeNode* pTreeNode, char* pCodeword );

// Insert a node into an ordered list of huffman nodes (in increasing frequency order)
struct huffmanListNode* huffmanListInsert( struct huffmanListNode* pInsertNode,
50
                                           struct huffmanListNode* pHead );

// Allocate memory for a Huffman tree node
struct huffmanTreeNode* huffmanTreeNodeNew();

// Deallocate memory for a Huffman tree node
void huffmanTreeNodeDelete( struct huffmanTreeNode* pTreeNode );

```

```
// Allocate memory for a Huffman list node  
struct huffmanListNode* huffmanListNodeNew( struct huffmanTreeNode* pTreeNode );
```

60

```
// Deallocate memory for a Huffman list node  
void huffmanListNodeDelete( struct huffmanListNode* pListNode );
```

```
#endif
```

D.7 error.c

```
//  
// Implementation of a text mimicry algorithm to pass covert data  
// Undergraduate thesis  
// Adam Tenenbaum, Engineering Science 0T2  
//  
// error.c – error related functions  
//
```

```
#include <stdio.h>
```

10

```
// Print out an error message on stderr  
void error( char* errmsg ) {  
    fprintf( stderr, "ERROR: %s\n", errmsg );  
}
```

D.8 error.h

```
//  
// Implementation of a text mimicry algorithm to pass covert data  
// Undergraduate thesis  
// Adam Tenenbaum, Engineering Science 0T2  
//  
// error.h - error related functions  
//
```

```
#ifndef __ERROR_H__
```

```
#define __ERROR_H__
```

10

```
// Print out an error on stderr  
void error( char* errmsg );
```

```
#endif
```

D.9 bool.h

```
//
// Implementation of a text mimicry algorithm to pass covert data
// Undergraduate thesis
// Adam Tenenbaum, Engineering Science 0T2
//
// bool.h - Definition of boolean type
//

#ifndef __BOOL_H__
#define __BOOL_H__                                10

typedef enum { FALSE = 0, TRUE = 1 } BOOL;

#endif
```

D.10 freqdist.c

```
//
// Implementation of a text mimicry algorithm to pass covert data
// Undergraduate thesis
// Adam Tenenbaum, Engineering Science 0T2
//
// freqdist.c - Calculates the alphabetical frequency distribution in
//              a file

#include <stdio.h>
#include <stdlib.h>                                10
#include <ctype.h>

// Main program loop
int main(int argc, char *argv[])
{
    FILE* fp; // File to be opened
    ulong frequency[26];
    ulong nTotalChars = 0;
    int i;

    for( i = 0; i < 26; i++ ) {                    20
        frequency[i] = 0;
    }

    printf( "-----\n" );
    printf( "  freqdist                \n" );
    printf( "  by Adam Tenenbaum          \n\n" );
    printf( "  Calculates the alphabetical frequency distribution \n" );
    printf( "  in a text file \n\n" );
    printf( "  usage: freqdist <input file> \n" );      30
    printf( "  e.g. freqdist message.txt \n" );
    printf( "-----\n\n" );
```

```

if( argc < 2 )
{
    printf("Invalid number of command-line arguments.\n");
    return( -1 );
}

fp = fopen( argv[1], "r" );
                                                                    40

if( !fp ) {
    printf("Main: Could not open file %s\n", argv[1] );
    exit( -1 );
}

while( !feof( fp ) ) {
    char c = fgetc( fp );

    if( c >= 'A' && c <= 'z' ) {
        char cUpper = toupper( c );
                                                                    50

        frequency[cUpper - 'A']++;
        nTotalChars++;
    }
}

fclose( fp );

// Done reading file: show output
                                                                    60

printf("Frequency Distribution Table\n");
printf("-----\n");

for( i = 0; i < 26; i++ ) {
    printf("%c: %.2f\n", 'a' + i,
        (double) frequency[i] / (double) nTotalChars * 100.0 );
}

return( 0 );
                                                                    70
}

```
