Abstract

Steganography provides for the embedding of information in a block of host data in conditions where perceptible modification of the host data is intolerable. Steganographic techniques are highly dependent on the character of the host data; a technique for embedding information in images might make subtle changes in hue, while a method for embedding information in audio data could exploit the limitations of the human ear by encoding the encapsulated information in inaudible frequency ranges. Current implementations of textual steganography exploit tolerances in typesetting by making minute changes in line placement and kerning in order to encapsulate hidden information, making them vulnerable to simple retypesetting attacks. This paper defines a framework for *lexical* steganography and discusses the details of an implementation.

1 Introduction

The field of steganography arose from the problem of data hiding; concealing information covertly by making it appear to be something else. With the maturation of steganography, commercial enterprises have arisen, seeking to provide the practical benefits of steganography—audio and image watermarking for copyright identification, authentication, and copy protection and control of digital video—to the public. Playboy Enterprises, publisher of the popular magazine, is reported to use a JPEG watermarking scheme by Digimarc in an effort to protect the copyright on its online properties.

Considering the number of people claiming to read *Playboy* "for the articles," one could conclude that the importance of the watermarking and protection of *text* should be considered at least as important as that of images, audio, and video. Unfortunately, textual steganography is not yet at the point where it can be used in the same manner as video and audio-based steganography is used today. If one wants to tag a block of text with some external identifier, one is either asked to include some obviously-superfluous chunk of text from a mimicry process in one's document[1], or else use a scheme such as that described in [3], involving

[1]Pages 247–263 of [2] describe a system for transforming any sequence of bits into a seemingly-innocuous block of text; unfortunately this block of text is in the form of a baseball game announcer's patter and is unlikely to fit in seamlessly with most documents.

the exploitation of visual tolerance in typography to encode information in the printed form of a document without affecting the underlying text.

None of these schemes, however, is suitable for digital media; as [1] correctly observes, "Soft-copy text is in many ways the most difficult place to hide data." Typographical watermarking is by nature prone to the "type-it-in-and-print-it-out" attack (the so printed version will not have the same minute typographical quirks as the original), and appended output from a steganographically-driven mimicry process is easily removed. What is needed is a system that can encode blocks of "soft" text with arbitrary information in a distributed manner with minimal disturbance to the underlying prose. I have created a framework for and trial implementation of such a system.

2 Framework for Lexical Steganography

The general data-hiding operation can be roughly described as the job of finding some watermarked entity S *equivalent* to some original entity E, but having the property that a *decoding function* $d()$ can extract some given watermark w from the watermarked entity S. If $s()$ is the watermarking function, then

$$s(E, w) = S \text{ where } S \in \{\text{entites equivalent to } E\} \text{ and } d(S) = w$$

The definitions of equivalence and the decoding function drive that of the encoding function, which need only produce some stego-entity to satisfy the dual requirements of "equivalence" to the original entity and the encoding of the watermark. These definitions are highly domain dependent—an image-based steganographic system will use a different encoding scheme than an audio-based one.

Perhaps equivalence in the visual domain can be defined as "looking the same, within a certain tolerance." How do we define "equivalence" in the domain of words? If we also define it as "looking the same, within a certain tolerance", then our decoding and encoding functions are likely to operate along the axis of appearance, and we shall end up with a scheme such as that described in [3]. Suppose, given our focus on "soft text" and the goal of avoiding the type-it-in-and-print-it-out attack (which is likely to separate the content of a work from its form), we define "equivalence" to be "giving the same basic impression to the reader, within a certain tolerance".

This strongly suggests an encoding/decoding function pair along the axis of *meaning.* The task, then, is in finding degrees of freedom in some textual entity that allow an underlying work to be changed while preserving its meaning. The job may be further rephrased by asked the question, "Once an *idea* is formulated (that is, an intended meaning), what further *decisions* are made in the transcription of this idea to the lexical medium?" For our purposes, any decision in the authoring process made *after* the formulation of an intended meaning can be considered a degree of freedom.

Clearly, the most major decisions being made when an idea is being realized into some language-based medium of communication are those of word choice. If the reader does not care whether a basement is *dank* or *clammy*, whether a main character's accomplishment is *praiseworthy* or *laudable*, or whether a knight is *chivalrous* or *gallant*, than these sorts of word choices can be used to encode information. This is the basic idea behind my implementation of lexical steganography.

3 The Naive Algorithm

Although previous techniques for applying steganography to the medium of text have focused on appearance, there has been some talk of lexical schemes. The largest discussion of the subject of which this author is aware can be found in [1]:

> A final category of data hiding in text involves changing the words themselves. Semantic methods are similar to the syntactic method. Rather than encoding binary data by exploiting ambiguity of form, these methods assign two synonyms primary or secondary value. For example, the word "big" could be considered primary and "large" secondary. Whether a word has primary or secondary value bears no relevance to how often it will be used, but, when decoding, primary words will be read as ones, secondary words as zeros ...
>
> Word webs such as WordNet can be used to automatically generate synonym tables. Where there are many synonyms, more than one bit can be encoded per substitution. (The choice between "propensity," "predilection," "penchant," and "proclivity" represents two bits of data.) Problems

> occur when the nuances of meaning interfere with the desire to encode data. For example, there is a problem with choice of the synonym pair "cool" and "chilly." Calling someone "cool" has very different connotations than calling them "chilly." The sentence "The students in line for registration are spaced-out" is also ambiguous.

What I call the naive algorithm for lexical steganography comes from these sorts of remarks. Consider an author, Alice, and a reader, Bob, who share a common database of 2^n-membered disjoint *synonym sets.* A synonym set is simply a ordered list[2] of words which are considered to have the same meaning and can be used interchangeably. The added restrictions are that each set must have number of members equal to some power n of 2, and thus the selection of a word from the synonym set can carry an integral n bits of information, and that the sets must be *disjoint*; that is, no word may appear in more than one synonym set.

When Alice wants to send some piece of information w secretly to Bob, she can embed it in some host manuscript E. The resulting stego-text S is sent to Bob, and to some eavesdropper it looks as innocuous as the original manuscript[3]. Bob, however, can run S through the decoding function, and extract the embedded data w.

The encoding function is defined as such:

> *For each word in the original text E, check if the word is in some synonym set. If so, then find how many elements are in the word's synonym set. Pull the log (base 2) of this number of bits off of the stack of bits, and change the word in question to that word that occupies the position in the synonym set equal to the number given (in binary) by the bits to be encoded.*

The decoding function is then simply:

> *For each word in the watermarked text S, check if the word is in some synonym set. If so, then find how many elements are in the word's synonym set. Find the position of the word in the synonym set, and add the binary equivalent of this number (with enough leading zeros added to have as many bits as the log (base 2) of the number of elements in the word's synonym set) to*

[2] If the elements do not already have some standard ordering, one can simply order them alphabetically.

[3] If Alice and Bob are using a standard synonym database, it may be trivial for an eavesdropper to read the stego-data. This is fine for watermarks, but for data hiding it would be wise to encrypt the embedded data prior to encoding.

the list of decoded bits.

This algorithm allows an arbitrary stream of bits to be embedded in some textual entity in a lexical manner. Unfortunately, there are a number of difficulties associated with it.

1. The linguistic model is simplistic; there is no explanation of how these sets of "interchangeable words" are to be derived. "Too" may be replaced with "also" sometimes, but in "This bed is too big," it clearly cannot be.

2. Each synonym set must correspond exactly to some integral number of bits. There is no ability to take advantage of having a 3-element set over a 2-element set, or even a 7-element set over a 4-element set.

3. The encoding is not adaptive. Where it is applied, the density of modifications is the same regardless of whether we have 10 bits to encode or 100.

4. There is no provision for choice in the encoding. If the author has a requirement that a particular word be used in a particular place (as opposed to whatever the watermark bits happen to dictate), there is no ability to compromise by making the change elsewhere in the document.

These four difficulties will each be addressed and, to differing degrees, solved by the implementation that is to be the subject of the rest of this paper.

4 The Linguistic Model and Synonym Database

I have developed, based primarily on data from WordNet, (described as "a computational linguist's dictionary"; see [4]) and moderately refined and augmented manually, a database of approximately 20 thousand words organized into interchangeability sets. To understand the construction of this database, it is necessary to understand somewhat WordNet's linguistic models, which is based on modern psycholinguistic ideas. Fortunately, the concepts of WordNet's organization are very similar to those which we have been discussing.

A lookup of a word results in a listing of all of the synonym sets to which it belongs; with one synonym set per *sense*. For instance, looking up the word "too" results in:

```
Overview of adv too

The adv too has 2 senses (first 2 from tagged texts)

1. excessively, overly, too -- (to an excessive degree; "too big")

2. besides, too, also, likewise, as well -- (in addition; "he has a Mercedes, too")
```

One sense of “too” is in the synonym set { excessively, overly, too }, and the other sense is in { besides, too, also, likewise, as well }. “Too” is not, therefore, “interchangeable” with “also”, even though they happen to be together in a synonym set.

Without attempting to guess which sense of a word is being used in our input, (see [5]), it should be possible to decide under exactly which conditions a set of words can be called “interchangeable” for our purposes. Consider three words **xxx**, **yyy**, and **zzz**:

Case one: **xxx** and **yyy** are both single-sensed, and they are in the same synonym set { **xxx**, **yyy** }. In this case, we clearly allow substitution of **yyy** for **xxx** under theoretically any circumstances (that is, for any sense used of either), and thus **xxx** and **yyy** are considered interchangeable.

Case two: **xxx** and **yyy** are doubly-sensed, as such:

$$S_1 : \{xxx, yyy\}$$

$$S_2 : \{xxx, yyy, zzz\}$$

Although **xxx** and **yyy** are doubly-sensed, we can still allow unconditional substitution between the two, since for every possible sense used of **xxx**, **yyy** is also in the corresponding synonym set. Therefore our set of interchangeable words is $S_1 \cap S_2$.

Case three: { **xxx**, **yyy** }, { **xxx**, **zzz** } No substitution can be made here for any word, since there is no word which is present in all of the synonym sets of any other word.

Case four: **xxx** and **yyy** are both single-sensed, and they are in the same synonym set { **xxx**, **yyy** }, but **yyy** has a space in it. Some WordNet elements are actually multi-word concepts with spaces in them; for the purposes of this implementation we ignore these and do not allow them to be the targets of substitutions.

Applying these criteria to WordNet's synonym sets allows approximately 30% of WordNet's 70,803 singleword entries to be included in some interchangeability set. The average number of words in a set is 2.56, the greatest is 13, and the smallest is 2. The distribution appears to be roughly inversely exponential.

5 Ideal Coding

The second problem associated with the naive algorithm is its inability to take advantage of the extra information-carrying capacity in regions of a synonym set beyond the highest power of 2. *Ideal Coding* allows us to take advantage of absolutely all of the information-carrying capacity in a document.

Imagine a document containing the words **aaa**, **mmm**, and **www**, where the relevant interchangeability sets involved (with a number assigned to each element) are: { **aaa**(0), **bbb**(1), **ccc**(2) }, { **mmm**(0), **nnn**(1), **ooo**(2), **ppp**(3), **qqq**(4) }, and { **www**(0), **xxx**(1), **yyy**(2) }. By replacing the locations containing **aaa**, **mmm**, and **www** with some member of their interchangeability set, we can achieve $3 \times 5 \times 3 = 45$ different states, which is enough to store 5 binary digits.

The mapping of these binary digits to states of the document is simple. By using the numbers tagged to each set element, can treat the document as a *multi-base number*—that is, its last digit signifies the "one's place" and can run from 0 through 2, its middle digit thus signifies the "threes place", and the first digit represents the "fifteens place". The standard counting sequence in binary thus maps logically to our "counting sequence" for document states as such:

Binary Stego-Digits		Mixed-Base State Code
00000	$\Rightarrow$	000
00001	$\Rightarrow$	001
00010	$\Rightarrow$	002
00011	$\Rightarrow$	010
$\vdots$		$\vdots$
01110	$\Rightarrow$	042
01111	$\Rightarrow$	100
$\vdots$		$\vdots$
11111	$\Rightarrow$	202

It is then a trivial operation to find the proper document state to match a desired coding.

6 The Word Choice Hash

Unfortunately, ideal coding has several disadvantages. One is that, with a typical document size, dealing with numbers of the size required can be computationally prohibitive. The average book-length work has about 1,500 bits of steganographic storage capacity—dealing with numbers on the order of 2^{1500} with special math routines is usually beyond the reach of most computer systems. Another disadvantage of ideal coding is the lack of adaptability—that is, the ability to scale down the impact on a document when the amount of information to be carried in a watermark is less than the steganographic storage capacity of a document. Liberation from both of these issues is provided by what I call the *word choice hash.* The word choice hash is a decoding function which maps a document's word choice state to a string of bits. It is designed such that the corresponding *encoding* function can be adaptive, near-ideal in efficiency, and not difficult to compute.

Two parameters control the operation of both the encoding and decoding versions of the word choice hash: the *oversample factor* and the *maximum frequency.* First the total information-carrying capacity of the document is computed, by summing the base-2 logarithms of the numbers of elements in each *zone*, where a zone is a location containing a word that is a member of an interchangeability set. This is used to determine the *modulation frequency*, which is an integer between 1 and the specific maximum frequency, inclusive.

The zonespace is divided into blocks, each of which is given responsibility for encoding a block of the watermark space with a size equal to the oversample factor. The size of each block of zonespace is determined by finding the back-to-back sequence of the smallest possible blocks (starting from the beginning), each containing enough steganographic information-carrying capacity to encode a number of bits equal to the oversample factor (which is the number of bits to be encoded by the block) multiplied by the modulation frequency.

Within each block, a variant of ideal coding is used to encode the corresponding bits of watermark. Since the capacity of the block will be greater than the amount of information assigned to it by a factor equal to the modulation frequency, we can assign the same hash value to multiple configurations of the block, and

Original text: b c = 1 2
Modulation frequency: 2
Bits assigned: 1 0
Zone 1: { a(0), b(1), c(2), d(3) }
Zone 2: { a(0), b(1), c(2), d(3) }

Binary Stego-Digits		Mixed-Base State Code
00	⇒	00
01	⇒	01
10	⇒	**02**
11	⇒	03
01	⇒	10
10	⇒	**11**
11	⇒	12
00	⇒	13
10	⇒	20
11	⇒	21
00	⇒	22
01	⇒	23
11	⇒	30
00	⇒	31
01	⇒	32
10	⇒	33

Figure 1: A simple word choice hash block. The bit capacity is greater than the number of bits assigned by a factor of the modulation frequency. The desired code of 10 can be achieved either by changing the first word to a(0), or by changing the second word to b(1). The author can be given a choice as to which path to take.

pick the configuration that is most true to the original document[4]. When the code values repeat, we skip a value in order to discourage falling "in sync" with the document state.

By creating redundancy through the use of higher frequencies when less information needs to be encoded, the word choice hash can be adaptively applied; with its impact on a document scaling with the amount of watermark information. (The maximal frequency parameter mainly serves as a restriction on computational requirements, because the complexity of the algorithm goes exponentially with the modulation frequency.) Using divided blocks with oversampling, we can get results that are almost as efficient as those that we get from ideal coding, but take much less time to encode.

[4]In the sample implementation, the "most true" metric is simply the number of words changed in the block, but a more sophisticated metric that attempted to match word usage statistics to those of the original document would not be difficult to implement.

7 Results

Lexical steganography works. My sample implementation uses a fixed Huffman code based on English language statistics from [6]. The steganographic information carrying capacity for normal English ASCII-encoded text appears to be near one-half of a percent. While comprehensive perception testing has not yet been done, encoded output appears to be well within what is tolerated for most prose. Source code for my sample implementation can be found in the appendices.

8 Future Directions

- Currently, the implementation of the decoder operates by "tuning through" the frequency spectrum[5] from 1 to the maximal frequency. With better pattern recognition and perhaps a slight degradation in signalling efficiency in response for more redundency, it could certainly be optimized to automatically "lock on" to the right frequency.

- Speed of implementation is still a problem. It appears that the most reasonable values for the oversample and maximum frequency constants are 4 and 4. With a more efficient implementation, it would be possible to achieve even better signalling efficiency.

- More research needs to be done into the best ways of avoiding periodicity when repeating code sequences within the block-state space. Cycling the order of the code, as now implemented, appears to work reasonably well but is clearly not ideal.

- The word choice hash could be optimized based on word usage information to assign more redundent codes to better choices for words.

9 Acknowledgements

I would like to thank Bill McCloskey for assistance in classifying the cases of synonym set overlap in section 4.

[5]With a high enough oversampling factor, the word choice hash at a frequency of 1 can be arbitrarily as good as ideal coding.

References

1. Bender at al., *Techniques for Data Hiding*, IBM Systems Journal v. 35 no. 3–4(1996), 313–336

2. Wayner, Peter, *Disappearing Cryptography*, Academic Press 1996

3. Brassil et al., *Hiding Information in Documents Images*, Conference on Information Sciences and Systems (CISS-95), March 1995

4. Beckwith et al., *Implementing a lexical network*, International Journal of Lexicography 3 (4), 1990, 302–312

5. Mihalcea and Moldovan, *Word Sense Disambiguation based on Semantic Density*, In Proceedings of the COLING/ACL Workshop on Usage of WordNet in Natural Language Processing Systems, Montreal, 1998

6. Harris, Mary Dee, *Introduction to Natural Language Processing*, Prentice Hall 1985

7. Winstein, Keith *Tyrannosaurus Lex—An Implementation of Lexical Steganography*, `http://www.imsa.edu/~keithw/tlex`

8. Sorkin, Andrew R., *Playboy Plans to Use Digital 'Watermarks'*, The New York Times, June 30, 1997

10 Appendix A: Sample Encoder Implementation

```
#!/usr/bin/perl -w

$oversample = 4;
$max_mode = 4;

open DATA, 'tlex.data';;
while (<DATA>) {
  my ($word, @synset) = split;
  $wordcache{ $word } = [ $word, @synset ];
}
close DATA;

%huffman = ( ' ' => "111",
             E => "000",
             T => "1101",
             A => "1011",
             I => "1001",
             O => "1000",
             R => "0111",
             S => "0110",
             N => "0100",
             H => "11001",
             C => "10101",
             L => "10100",
             D => "01011",
             M => "00111",
             U => "00110",
             P => "00100",
             F => "110001",
             G => "110000",
             B => "010100",
             W => "001011",
             Y => "001010",
             V => "0101010",
             K => "01010110",
             X => "010101110",
             Q => "0101011110",
             J => "01010111110",
             Z => "01010111111"
           );

my $bitstring;

for (split '', shift) { $bitstring .= $huffman{ uc $_ } }

sub nextbit {
  return(undef) if ($bitstring eq '');
  my $x = substr($bitstring, 0, 1);
  unless (length $bitstring == 0) {
    $bitstring = substr($bitstring, 1);
  } else {
```

```
    $bitstring = '';
  }
  return $x;
}

my $line_num = -1;
my @nonwordrec;
my %synsetable;
my %ordinal;
my $s = 0;

while (defined(my $line = <>)) {
  $line_num++;

  chomp $line;
  $line = 'a ' . $line;

  my @words = split /[^A-Za-z\-]+/, $line;
  my @nonwords = split /[A-Za-z\-]+/, $line;

  $wordrec[ $line_num ] = \@words;
  $nonwordrec[ $line_num ] = \@nonwords;

  my $word_num = -1;
 WORD: for my $word (@words) {
    $word_num++;
    next WORD unless (defined($wordcache{ $word }));
    $synsetable{ $line_num . ' ' . $word_num } = $wordcache{ $word };
    $ordinal{ $line_num . ' ' . $word_num } = $s++;

    print STDERR "Looking at $word (", (join ", ", @{$wordcache{ $word }})
      , ") \n";
  }
}

my $variance = 0;
for (keys %synsetable) {
  print STDERR "Eyeing $_\n";
  $variance += (log (scalar @{$synsetable{ $_ }}))/(log 2);
}

print STDERR "Length of message: ", length $bitstring, "\n";
print STDERR "Variance bits in text: $variance\n";

my $mode = int($variance / length $bitstring);

if ($mode > $max_mode) {
  $mode = $max_mode;
}

unless ($mode >= 1) {
  print STDERR "Sorry, there is not enough entropy to encode the message.\n";
  exit 5;
```

```
}

print STDERR "Using mode $mode encoding frequency.\n";

my @zones = sort { $ordinal{ $a } <=> $ordinal{ $b } } keys %synsetable;

while (length $bitstring > 0) {
  print STDERR "String: $bitstring, Length: ", length $bitstring, "\n";
  my @bit_array;
  for (1 .. $oversample) {
    my $bit = nextbit;
    last unless (defined $bit);
    push @bit_array, $bit;
    print STDERR "pushed: $bit_array[ $_ - 1 ] ", scalar(@bit_array), "\n";
  }
  my $bits_assigned = 0;
  my @these_zones;
  while ($bits_assigned < $mode * scalar(@bit_array)) {
    push @these_zones, (my $z = shift @zones);
    $bits_assigned += (log(scalar @{$synsetable{ $z }}))/(log 2);
  }
  encode(\@bit_array, \@these_zones);
}

sub encode {
  my ($r_bits, $r_zones) = @_;

  my (@bits) = @{$r_bits};
  my (@zones) = @{$r_zones};

  my $num_bits = scalar @bits;

  my @choices;
  my @word;

  print STDERR "Encoding @bits into ", (join ", ", @zones), "\n";

  for my $i (0 .. $#zones) {
    my ($l, $w) = split /\s+/, $zones[$i];
    $word[$i] = $wordrec[ $l ][ $w ];
    @{$choices[$i]} = sort @{$wordcache{ $word[$i] }};
  }

  print STDERR "word: @word\n";

  print STDERR "*\n";
  my @vector;
 WORD: for my $i (0 .. $#word) {
    for my $j (0 .. (scalar @{$choices[$i]} - 1)) {
      print STDERR "-";
      if ($choices[$i][$j] eq $word[$i]) {
        push @vector, $j;
        next WORD;
```

```
    }
  }
}

my @iterarray = (0) x (scalar @word);
my @codearray = (0) x $num_bits;
my @codes;

my $iter = 0;

print STDERR "^\n";
while (scalar @iterarray == scalar @word) {
  $iter++;
  print STDERR "." unless ($iter % 1000);

  my $equal = 1;
  for (my $q = 0; $q <= $#bits; $q++) {
    $equal = 0 unless ($codearray[$q] eq $bits[$q]);
  }

  if ($equal) {
    push @codes, (join '', @iterarray);
  }

  $iterarray[ 0 ]++;
  $codearray[ 0 ]++;
  for my $i (0 .. $#iterarray) {
    if ($iterarray[ $i ] >= scalar @{$choices[$i]}) {
      $iterarray[ $i ] = 0;
      $iterarray[ $i + 1 ]++;
    }
  }

  if (scalar @codearray > $num_bits) {
    pop @codearray;
    $codearray[ 0 ]++;
    for my $i (0 .. $#codearray) {
      if ($codearray[ $i ] >= 2) {
        $codearray[ $i ] = 0;
        $codearray[ $i + 1 ]++;
      }
    }
  }
  for my $i (0 .. $#codearray) {
    if ($codearray[ $i ] >= 2) {
      $codearray[ $i ] = 0;
      $codearray[ $i + 1 ]++;
    }
  }
}

print STDERR "bits: {", join '', @bits, "} \n";
```

```
  print STDERR "finding best in ", scalar @codes, " encodings ...";

  my $best = v_dist(\@vector, $codes[0]);
  my $best_num = 0;
  my $itera = 0;
  for (@codes) {
    if (v_dist(\@vector, $_) < $best) {
      $best = v_dist(\@vector, $_);
      $best_num = $itera;
    }
    $itera++;
  }
  print STDERR "best_num: $best_num\n";
  my @enc_choices = split '', $codes[$best_num];
  print STDERR "done\n";
  print STDERR "choices: @enc_choices (",
  v_dist(\@vector, $codes[$best_num]), ")\n";

  for my $i (0 .. $#enc_choices) {
    my ($l, $w) = split /\s+/, $zones[$i];
    $wordrec[ $l ][ $w ] = $choices[$i][ $enc_choices[$i] ];
  }
}

sub v_dist {
  my ($vec_ref, $p) = @_;
  my ($dist) = (0, 0);

  for $i (0 .. (scalar @{$vec_ref} - 1)) {
    unless ( (@{$vec_ref})[$i] eq substr($p, $i, 1) ) {
      $dist++;
    }
  }
  return ($dist);
}

for my $i (0 .. $line_num ) {
  my $line = '';
  *words = $wordrec[ $i ];
  *nonwords = $nonwordrec[ $i ];
  for (my $i = 0; $i < scalar @words + scalar @nonwords; $i++) {
    $line .= ($i % 2 ? $words[($i - 1) / 2] : $nonwords[$i / 2]);
  }
  $line = substr ($line, 2);
  print $line, "\n";
}
```

11 Appendix B: Sample Decoder Implementation

```
#!/usr/bin/perl -w

$oversample = 4;
$max_mode = 4;

open DATA, 'tlex.data';;
while (<DATA>) {
  my ($word, @synset) = split;
  $wordcache{ $word } = [ $word, @synset ];
}
close DATA;

%huffman = ( ' ' => "111",
             E => "000",
             T => "1101",
             A => "1011",
             I => "1001",
             O => "1000",
             R => "0111",
             S => "0110",
             N => "0100",
             H => "11001",
             C => "10101",
             L => "10100",
             D => "01011",
             M => "00111",
             U => "00110",
             P => "00100",
             F => "110001",
             G => "110000",
             B => "010100",
             W => "001011",
             Y => "001010",
             V => "0101010",
             K => "01010110",
             X => "010101110",
             Q => "0101011110",
             J => "01010111110",
             Z => "01010111111"
           );

%revhuff = reverse %huffman;

my $line_num = -1;
my @nonwordrec;
my %synsetable;
my %ordinal;
my $s = 0;

while (defined(my $line = <>)) {
  $line_num++;
```

```
  chomp $line;
  $line = 'a ' . $line;

  my @words = split /[^A-Za-z\-]+/, $line;
  my @nonwords = split /[A-Za-z\-]+/, $line;

  $wordrec[ $line_num ] = \@words;
  $nonwordrec[ $line_num ] = \@nonwords;

  my $word_num = -1;
 WORD: for my $word (@words) {
    $word_num++;
    next WORD unless (defined($wordcache{ $word }));
    $synsetable{ $line_num . ' ' . $word_num } = $wordcache{ $word };
    $ordinal{ $line_num . ' ' . $word_num } = $s++;

    print STDERR "Looking at $word (", (join ", ", @{$wordcache{ $word }})
      , ") \n";
  }
}

my $variance = 0;
for (keys %synsetable) {
  print STDERR "Eyeing $_\n";
  $variance += (log (scalar @{$synsetable{ $_ }}))/(log 2);
}

print STDERR "Variance bits in text: $variance\n";

my @real_zones = sort { $ordinal{ $a } <=> $ordinal{ $b } } keys %synsetable;

print STDERR "rz: @real_zones\n";

for my $mode (1 .. $max_mode) {
  print STDERR "Using mode $mode encoding frequency.\n";

  my (@zones) = @real_zones;
  $bits = '';

  BIG: while (scalar @zones > 0) {
    my @these_zones;
    my $bits_assigned = 0;
    while ($bits_assigned < $mode * $oversample) {
      if (scalar @zones == 0) {
        last BIG;
      }
      push @these_zones, (my $z = shift @zones);
      print STDERR "z: $z\n";
      $bits_assigned += (log(scalar @{$synsetable{ $z }}))/(log 2);
    }
    print STDERR "tz: @these_zones\n";
    decode(\@these_zones, $oversample);
```

```
  }
  print "bits: $bits\n";
  compute($bits);
}

sub compute {
  my ($bits) = @_;
  my @bitarray = split '', $bits;

  my $token = '';

  while (scalar @bitarray > 0) {
    $token .= shift @bitarray;
    if (defined($revhuff{ $token })) {
      print $revhuff{ $token };
      $token = '';
    }
  }
  print "\n";
}

sub decode {
  my ($r_zones, $num_bits) = @_;

  my (@zones) = @{$r_zones};

  my @choices;
  my @word;

  for my $i (0 .. $#zones) {
    my ($l, $w) = split /\s+/, $zones[$i];
    $word[$i] = $wordrec[ $l ][ $w ];
    @{$choices[$i]} = sort @{$wordcache{ $word[$i] }};
  }

  print STDERR "word: @word\n";

  print STDERR "*\n";
  my @vector;
  WORD: for my $i (0 .. $#word) {
    for my $j (0 .. (scalar @{$choices[$i]} - 1)) {
      print STDERR "-";
      if ($choices[$i][$j] eq $word[$i]) {
        push @vector, $j;
        next WORD;
      }
    }
  }

  my @iterarray = (0) x (scalar @word);
  my @codearray = (0) x $num_bits;
  print STDERR "num_bits: $num_bits\n";
  my @codes;
```

```
  my $iter = 0;

  print STDERR "^\n";
  LOOP: while (scalar @iterarray == scalar @word) {
    $iter++;
    print STDERR "." unless ($iter % 1000);

    my $equal = 1;
    for (my $q = 0; $q <= $#iterarray; $q++) {
      $equal = 0 unless ($iterarray[$q] eq $vector[$q]);
    }

    if ($equal) {
      $bits .= (join '', @codearray[0 .. ($num_bits - 1)]);
      print STDERR "bits now $bits\n";
      last LOOP;
    }

    $iterarray[ 0 ]++;
    $codearray[ 0 ]++;
    for my $i (0 .. $#iterarray) {
      if ($iterarray[ $i ] >= scalar @{$choices[$i]}) {
        $iterarray[ $i ] = 0;
        $iterarray[ $i + 1 ]++;
      }
    }

    if (scalar @codearray > $num_bits) {
      pop @codearray;
      $codearray[ 0 ]++;
      for my $i (0 .. $#codearray) {
        if ($codearray[ $i ] >= 2) {
          $codearray[ $i ] = 0;
          $codearray[ $i + 1 ]++;
        }
      }
    }
    for my $i (0 .. $#codearray) {
      if ($codearray[ $i ] >= 2) {
        $codearray[ $i ] = 0;
        $codearray[ $i + 1 ]++;
      }
    }
  }
}
```