PRIME-BASED MIMIC FUNCTIONS

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Wesley J. Connell, B.S.

San Marcos, Texas
August 2009

ACKNOWLEDGEMENTS

Dan Tamir, PhD.
Carol Hazlewood, PhD.
Mina Guirguis, PhD.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

PRIME-BASED MIMIC FUNCTIONS

by

Wesley J. Connell

Texas State University-San Marcos

August 2009

SUPERVISING PROFESSOR: DAN TAMIR

Many steganography techniques suffer from a low secret-to-cover ratio or are vulnerable to statistics-based steganalysis. Prime-based mimic functions provide an efficient and substantially covert method to hiding information for steganography. Experiment results show a competitive secret-to-cover ratio and low language model perplexities.

1. Introduction

Steganography is the art and science of hiding information in plain sight. By hiding information in plain sight, such as images, audio, or text, we are sending information over a *covert* channel to completely avoid detection. In fact, almost any abstract medium can be utilized to provide cover for the information to be concealed. This abstract medium is referred to as the *cover medium*. In order to perform optimally, the cover medium must be chosen and utilized in such a manner that neither human nor machine can recognize it as a cover for hidden information. Unfortunately, with many steganography techniques, the size ratio of secret message to the cover is quite low. Additionally, these techniques are vulnerable to automated steganalysis.

In this thesis, we define a prime-based mimic function that encodes a secret message into the sentences of a given grammar such that the sentences are statistically similar to typical sentences in the grammar while maintaining a competitive bit-rate. We show this by implementing a prime-based mimic function and statistically analyze the generated sentences with a language model that calculates geometric and average perplexities. Additionally, the average bit-rate is calculated and compared to other steganography techniques. Our results show low perplexities and competitive secret-to-cover ratios. However, we begin with a literature survey of steganography research.

2. Literature Survey

In this section we shall discuss popular steganography research related to image steganography, audio steganography, timing-based covert channels, and mimic functions along with technique robustness and fundamental steganography concepts.

Originating with the Greeks, history shows a multitude of steganography techniques. For instance, in 440 BC, Histiaeus used the tattooing of a trusted scout's head, later covered by hair, to conceal a message [1]. Later, during WWII, additions to the field would include the use of microdots and invisible inks [1].

As described in the introduction and shown through the historical examples, the purpose of steganography is to send information over a *covert* channel, thus avoiding the detection of the secret message altogether. Similar to the scenarios used in research and examples of encryption, we shall also use the actors Alice, Bob, and Eve to be our sender, receiver, and observer respectively. Typically, the knowledge held by Eve is complete with regards to anything transmitted and methodology used.

Our base scenario has Alice attempting to send a secret message to Bob using a *cover*. The cover is the medium in which the secret message is embedded, such as image data, audio data, and text data. Hopefully, Eve, having complete knowledge of the

steganography method and any additional transmitted information, such as sequence lengths or public-keys, cannot detect the presence of a secret message. Given Eve's complete knowledge, a strong steganography method would still allow the successful covert transmission of information between Alice and Bob.

2.1 Robustness

During the course of our base scenario, if Eve detects the presence of a secret message in the cover, then she has a choice to make: allow the cover to continue to be transmitted without modification or to attempt to modify the cover in such a manner as to prohibit Bob from reading the secret message. A steganography technique's resistance to such a modification is called its *robustness*. Obviously, Eve's decision is based on her intelligence priorities. For the purposes of detailing robustness, we assume Eve will attempt to modify the cover.

Instead of completely ruining the cover, Eve can subtly introduce noise, distortion, or another domain specific modification to maintain the cohesion and usefulness of the cover while rendering the secret message unusable by Bob. Domain specific modifications include applying filters to an image, re-sampling an audio file, and replacing words with synonyms in a story.

2.2 Types of Steganography

Three categories of steganography are recognized, each with varying levels of

knowledge transmission and assumption. The first, pure steganography, allows no prior exchange of prior information and assumes both the sender and receiver know the encoding and decoding algorithm. Unfortunately, since we assume Eve has complete knowledge of the steganography system, then she also has the ability to decode the secret message. The second, secret key steganography, involves the transmission of a secret key that is required to decode information in the cover. Again, we can assume Eve has intercepted the secret key and can now decode the secret message. The third, public-key steganography, uses a mechanism similar to public-key encryption to transmit a public-key that can be used to encode the secret message. Fortunately, even though Eve has knowledge of this public-key, she cannot decode the message.

2.3 Language Models

Given that most lexical and linguistic steganography techniques attempt to acquire the statistical properties of a defined language, it is appropriate to analyze their output. An efficient method for doing so involves the use of *language models*. Language models, using the most simplistic definition, build a probability distribution for the words and relationships in a body of text, or corpus. This corpus serves as the training data set for the model and should contain the statistical properties that the cover wishes to acquire. Many types of language models exist and have been well-researched. However, the most popular type of language model is an *n-gram* based model that segments the words of the corpus into 1, 2, … , n-tuples and calculates their conditional probabilities. This type of language model is the default model used in our experiment.

2.4 Perplexity

Amongst the numerous measurements performed within a language model, the most interesting to us is *perplexity*. Perplexity is defined as two raised to the power of the entropy of the random variable X, or,

$$2^{-\sum_{i=1}^{n} p(x_i)\log_2(p(x_i))},$$

where n is the number of events in X and $p(x_i)$ is the probability of event x_i occurring [2]. A more intuitive definition of perplexity is the measurement of how surprised a language model is to see a specific sample. The lower the total entropy of the sample, the lower the uncertainty of the sample, and the lower the perplexity of the sample. Raising the entropy by a power of two has a normalizing effect on the logarithm, but still maintains the proper proportionality of having a lower perplexity being equivalent to less surprise. Fortunately, the perplexity is automatically calculated for us in our experiment by SRILM toolkit [3].

2.4 Image

Image data serve as an excellent cover medium due to their variety in compression algorithms, a high bit-depth, and popularity in use. Modern compression algorithms, such as the JPEG algorithm, allow for a reasonable amount of noise to be introduced into the image data. This noise, coupled with a high bit-depth can itself be a secret message. The popularity in use allows a small stream of image data, serving as a covert channel, to be hidden amongst the many other streams of legitimate image data.

Two popular steganography techniques, Least Significant Bit Substitution and Domain Transformation, both operate on image data [1]. In order to express the approximate bandwidth of these techniques, a sample of the first 100 JPEG-encoded images from an image search on Google were measured. Since each image was restricted to a resolution of 640x480, or 307,200 pixels, the search yielded an average size of 87 kilobytes. We define an image steganography technique's bandwidth to be,

$$b = \frac{p}{q} r \quad ,$$

where p is the number of secret bits encoded into the cover, q is the total size, in bits, of the cover, and r is the bandwidth of the channel the cover is transmitted on. For our purposes, we will assume r is 1.544 Mbit/sec, or a T1 communications line. Additionally, q is the average size yielded by the image search, or 712,704 bits.

2.4.1 Least Significant Bit Substitution

When examining an image, having a bit-depth greater than two and a color channel-based encoding scheme, the gradient difference between sequential values is slight. At higher bit-depths, the difference can be completely indistinguishable. Taking advantage of this difference, Least Significant Bit Substitution (LSB) replaces a constant number of bits from each pixel or color channel with the same number of bits from a secret message [1]. Spread over the entire set of image data, the entire secret message can encoded while inserting a minimal amount of noise into the image data. Unfortunately, this technique, although popular and easy to implement, suffers from quick detection by

standard steganalysis techniques, particularly Laplace Filtering. Additionally, because the secret message is stored in the values, rather than relationships, of the pixels, LSB cannot resist modification of the cover well. The trade-off for the lack of modification resistance is bandwidth. Given a a single-bit bit substitution and a sample image from the search above, we can encode 307,200 bits, or 38 kilobytes; this figures to a secret-to-cover ratio of approximately 0.43 and a bandwidth of 633 Kbit/sec.

2.4.2 Domain Transformation

Another popular steganography technique, Domain Transformation, involves transforming the image data from the time domain into the frequency domain. The raw pixel values can easily be mapped to a time series with each pixel value k representing sample n, where N is the total number of samples or pixel values. The transformation, commonly called the Discrete Fourier Transform (DFT), returns the frequency components of the image data. The DFT is defined as

$$S(k)=\mathrm{F}\{s\}=\sum_{n=0}^{N-1} s(n)\exp(-\frac{2in\pi k}{N}) \quad ,$$

where i is the imaginary unit $\sqrt{-1}$ [1]. In order to return the frequency components back into the time domain, the inverse DFT must be performed. This is defined as,

$$S(k)=\mathrm{F}^{-1}\{S\}=\sum_{n=0}^{N-1} S(n)\exp(\frac{2in\pi k}{N}) \quad .$$

With the help of the DFT, a domain transformation can be used to encode the secret message within image data. First, the image data are transformed into the

frequency domain using the DFT. Next, a subtle modification can be performed according to a previously arranged method. For instance, the magnitude of a frequency component can be increased or decreased in order to represent a one or a zero respectively [1]. Finally, the frequency components are transformed back into the image data and transmitted. Upon receipt of the image data, the process is reversed to retrieve the modification made to the frequency and the bit is decoded.

A steganography technique that utilizes a domain transformation can be quite robust. Because the frequency components themselves contain the secret message, any change to another image format, such from JPEG to PNG has no effect on the components. Additionally, many modifications such as filters will not alter the frequency components enough to ruin the secret message. Unfortunately, the transmission rate using domain transformation can be low compared to the LSB technique since domain transformation requires that the DFT be performed on a block of data. For instance, the JPEG image format uses a 8x8 pixel block of data to perform its compression. The same block size would be a suitable block size for the DFT as well. Thus, a 640x480 resolution image could only encode 600 bits of information. Thus, the secret-to-cover ratio for a domain transformation is approximately 0.00084 with a bandwidth of 1.3 Kbits/sec. Here, we have traded a high robustness for a low bit-rate.

2.4.3 Laplace Filtering

Image steganography techniques, such as LSB, which introduce noise may be

susceptible to steganalysis attacks involving Laplace Filtering. Primarily used in physics to model wave propagation and heat flow, this filtering attack utilizes the Laplace operator ∇^2 to detect noise within an image [1]. The following equation is evaluated for each pixel in the suspect image:

$$\nabla^2 p(x,y)=p(x+1,y)+p(x-1,y)+p(x,y+1)+p(x,y-1)-4\mathrm{p}(x,y) \quad .$$

In this equation, the function p represents the value of the pixel at coordinate (x,y). The resulting histogram from the calculations show a sharp spike centered around zero with a tightness proportional to the amount of noise present in the image. If a wide and broken spike is found, the image has considerable amounts of noise and may have been processed with a steganography technique. Additionally, varying sizes of pixel blocks can be used in the Laplace filter, such as pairs of pixels or 8x8 blocks of pixels. Together, the resulting histograms may show broken spikes.

2.5 Audio

Audio data can also become the cover for a covert message. The techniques used in image steganography may be applied in the domain of audio steganography with varying results. Petitcolas [1] states that since human audio perception is extremely sensitive to noise, the steganography techniques which introduce noise into the cover, such as LSB Substitution and Domain Transformation, are easier to detect. Therefore, techniques which exploit weaknesses in human audio perception perform better.

Since the steganography techniques are intended to be used on digital data,

the audio data must be represented as a discrete time series. A common encoding scheme for a discrete time signal is Pulse Code Modulation, or PCM. This type of modulation normalizes the magnitudes of the signal to a discrete range, typically an 8-bit or 16-bit signed value. Any abnormally large spikes or small ripples in the signal are clipped at the maximum range or rounded down to the smallest range respectively. In order to express the bandwidth of the following techniques, we assume the digital audio data serving as the cover has been encoded using an 8-bit signed PCM scheme sampled at 8,192 KHz, suitable for voice communication, for ten seconds. The same equation used to calculate image bandwidth, found in section 3.4, applies to audio bandwidth with the exception that q is now 81,920 bytes or 655,360 bits.

2.5.1 Phase Coding

While human audio perception recognizes changes in noise levels quite well, it has a difficult time detecting phase shifts. The phase coding technique exploits this weakness by introducing a phase shift into the cover by performing a DFT on the cover, modifying the resulting phase matrix and performing the inverse DFT.

The cover c, is split into a series of N sequences, $c_i(n)$ of length $l(m)$ and a DFT is performed on the set of sequences. The result of the DFT gives us the phase matrix $\phi_i(k)$ and the transform magnitudes $A_i(k)$. These are found by the following functions,

$$A_t(k)=\sqrt{\mathrm{Re}[\mathrm{F}\{c_t\}(k)]^2+\mathrm{Im}[\mathrm{F}\{c_t\}(k)]^2}$$

and

$$\phi_t(k)=\arctan\left(\frac{\mathrm{Im}[\mathrm{F}\{c_t\}(k)]}{\mathrm{Re}[\mathrm{F}\{c_t(k)\}]}\right)$$

Now, in order to phase shift the cover data, we set the first element in the phase matrix to be a small multiple of π . Since Petitcolas [1] uses $\pi/2$, so shall we. To calculate the $\phi_t(k)$, the new phase matrices, let,

$$\tilde{\phi}_0(k)=\begin{cases}\frac{\pi}{2}, \text{if } m_k=0\\ -\frac{\pi}{2}, \text{if } m_k=1\end{cases}.$$

With this assignment, we phase shift 90 degrees ahead or back to embed a single bit. The remaining sequences will be calculated by performing the sum of the original phase differences and the previous element of the phase matrix. Thus,

$$\begin{aligned}\tilde{\phi}_1(k)&=\tilde{\phi}_0(k)+[\phi_1(k)-\phi_0(k)]\\ &\cdots\\ \tilde{\phi}_N(k)&=\tilde{\phi}_{N-1}(k)+[\phi_N(k)-\phi_{N-1}(k)]\end{aligned}.$$

Once the new phase matrix has been calculated, the inverse DFT is performed using $A_t(k)$ and $\tilde{\phi}_t(k)$ to produce the phase-shifted cover. The receiver, having knowledge of $l(m)$, can now use the DFT to retrieve the phase shift and determine the embedded bit. Knowledge of $l(m)$ is required since the DFT must operate on data blocks of uniform size.

An issue with phase coding is the extremely small data transmission rate of

the technique. For a single bit to be sent, an entire set of audio data must be transmitted and gives us an approximate 0.00001 secret-to-cover ratio and a bandwidth of 19 bits/sec. Additionally, a consideration must be given to precision in implementation since the DFT will require the use of floating-point arithmetic. Incorrect data could be sent or received if excessive loss of precision is not prevented or checked. In spite of the issues involved with phase coding, Chang [5] has shown phase coding to have robustness against re-sampling of the cover.

2.5.2 Echo Hiding

Another audio steganography technique involves the modification of the cover by inserting a variable echo periodically throughout the data. The variability of the echo which is inserted into the cover determines the bit-depth of the embedded data. For instance, Gruhl [5] uses Δt and $\Delta t'$, the time delay of the echo, to embed a single bit of data. With additional intervals of Δt, an increased bit-depth can be achieved with great precision and complexity in the encoding and decoding functions.

As in phase coding, the cover c must be split into N sequences of length $l(m)$. Each sequence can contain an inserted echo to represent a single datum. Katzenbeisser [4] gives,

$$c(t)=f(t)+\alpha f(t-\Delta t) \quad ,$$

as the general function to calculate the new cover data with the inserted echo, where α is a small constant less than one to represent a minor degradation of the echo

signal . By replacing Δt with $\Delta t'$, we can embed a different value.

Again, with the receiver having knowledge of the sequence length $l(m)$, the cover is processed using autocorrelation. A signal spike is present at the beginning of the echo and is thus the determination of the embedded value. The process of autocorrelation is a statistical process comparing different points in time and determines their correlation. This is found by

$$R(t,s)=\frac{E[(X_t-\mu_t)(X_s-\mu_s)]}{\sigma_t\sigma_s}$$

where X has mean μ and variance σ . When well-defined, R , inclusively falls between -1 and 1, or where -1 signifies complete non-correlation and 1 signifies perfect correlation respectively. For the purposes of echo hiding, the signal spike occurs when an echo has been encountered and correlated to the original signal producing a value close to 1.

Gruhl [6] has shown echo hiding to have a potentially higher data transmission rate than phase coding with an equivalent robustness. The transmission rate is potentially higher, since the secret-to-cover ratio is $N/81920$ with a respective bandwidth of 19N bits/sec, where N is the number of echo segments encoded. Obviously, if only one echo segment has been encoded, then the bit-rate is equivalent to phase coding. Additionally, Gruhl [6] details the typical steganalysis attack on the echo hiding technique. While it is possible to detect and modify a cover using the detailed

attack, it relies on brute force and is limited to a small range of values for Δt . Thus, the steganalysis can be easily overwhelmed with excessive transmission or using an obtuse range of values for Δt .

2.6 Timing-based techniques

The techniques shown for image and audio data rely on storing the secret message within the cover. A technique shown by Guirguis [7] and Cabuk [8], relies on the timing of network transmissions to encode a secret message. By forcing the loss of a network transmission at a specific interval predetermined by the parties, the receipt or loss of said transmission can represent a single bit. This type of technique can be quite stealthy with a trade-off to a comparably low bit-rate of 4-12 bits/sec [7].

2.7 Cover Generation and Mimic Functions

Steganography applications using image-based or audio-based systems are required to use a cover channel or medium independent of the payload. As discussed in [4], the use of an independent cover implies that consideration must be given to proper choice of cover; consequences of insecurity arise when a cover is poorly chosen. Since it is difficult for a human operator to examine even a small percentage of possible covers, automated systems are employed to search for the various statistical properties of covers. Holotyak [9] and Fridrich [10] are just two from a thorough body of research into statistical steganography.

2.7.1 Cover Generation

Given that a chosen cover can be detected using a statistical property inherent to the cover itself or the steganography technique employed, the chosen cover must have statistical properties unknown to the steganalyst or statistical properties that make it indistinguishable from non-cover data of the same medium. The first, statistical properties which are unknown to the steganalyst, is simply security through obscurity and hardly a reasonable choice. The second defines the goal of *cover generation*. Cover generation accepts a payload as usual, but causes the cover to be dependent on the payload in a manner that maintains desirable statistical properties.

2.7.2 Mimic Functions

In [11] and [12], Wayner defines mimic functions, a cover generation technique that uses text as a cover medium. Mimic functions begin with a secret message to encode and end with a generated cover consisting of a body of text which is accepted by a context-free grammar. Furthermore, the generated cover is shown by Wayner to have a possible strength, or resistance to steganalysis, proportional to the average complexity of the context-free grammar [12]. This technique forms the basis for our definition of prime-based mimic functions with the exception that the Huffman-coding portion of the technique is omitted.

A given secret message is first compressed using a Huffman-coding scheme [18]. This scheme compresses the secret message based upon the statistical properties of

the text so that the more frequent characters are represented by fewer bits. Once Huffman-coded, the resulting set of bits are used to determine the set of productions to expand in the context-free grammar. After the final expansion, the text may be transmitted as a steganography cover and decoded, using the inverse of the described process, to retrieve the secret message.

The strength of the transmitted message is proportional to the average complexity of the context-free grammar. This complexity is defined as

$$\prod_{i=0}^{n} p(a_i) 2^{E(t_i)} \quad ,$$

where $p(a_i)$ is the probability that terminal a_i appears in a string generated by the grammar and $E(t_i)$ is the entropy of a set of particular strings generated by the grammar. Wayner states, "the larger it is, the more secure the system may be against probabilistic attacks" [12].

Although not discussed in the literature, the robustness of mimic functions depends upon the redundancy within the context-free grammar and whether or not misspelled words and garbage characters would be accepted. The transmission rate, however, is higher than the typical image and audio based steganography techniques. Wayner provides an example which encodes, "Paul is dead! I am the Walrus! Buy something right now. Don't shoplift. Buy! Buy! Here are the plans to the Overthruster, Sergei. Yoyodyne forever." Assuming the message was stored as a standard 8-bit byte,

the example encodes 148 bytes into a 12,660 byte cover message yielding a secret-to-cover ratio of 0.012 and a bit-rate of 18 Kbits/sec.

2.7.3 NICETEXT

Another approach to the goal of using text for steganography, is the set of functions called NICETEXT [13]. NICETEXT utilizes a collection of dictionaries and styles to construct a cover that is statistically similar to a specific and defined language. The dictionaries are a combination of manually and automatically generated word-type and word pairs which assist in selecting an appropriate word for a sentence within the cover based on usage frequency within the language. The styles, composed of sequences of word-types, enforce a grammar within the cover by simulating a probabilistic context-free grammar. Thus, the dictionaries and styles intersect by word-type. Using a set of bits as input, a specific style is selected from a table keyed on the bit signature and the word-types replaced with dictionary words. Chapman's thesis [13] contains a clear example which encodes 88 bytes of data into a 2000 byte cover yielding a secret-to-cover ratio of 0.044 and an average bit-rate of 67 Kbits/sec.

3. Prime-based Mimic Functions

In this section we formally define a prime-based mimic function and give a trivial example of its usage. Supplemental to the prime-based mimic function, we also define the criteria for a required mapping algorithm and briefly discuss its limitations.

A prime-based mimic function modifies data to fit the statistical properties of a context-free grammar. The context-free grammar is modified by adding a function ξ which maps productions to a set of sequential prime numbers and 1. Let

$$G=(V,T,S,P,\xi) \ ,$$

where

V is a finite set of variables,
T is a finite set of terminal symbols,
$S \in V$ is the starting symbol,
P is a finite set of productions,
ξ is a function mapping productions to prime-numbers and 1.

Given the application of the Fundamental Theorem of Arithmetic [14], which states that any integer greater than one is composed of the product of a finite set of prime numbers, the inclusion of ξ allows a unique expansion of productions within the grammar to be representative of an integer. In other words, an integer can be encoded using sentences within a language described by G. For example, let

$G=(\{S\},\{a,b\},S,P,\xi)$, with productions

$$p_0 = S \rightarrow aSa$$
$$p_1 = S \rightarrow bSb$$
$$p_2 = S \rightarrow \lambda,$$

and $\xi = \{\{p_0, 2\}, \{p_1, 3\}, \{p_2, 1\}\}$. During the expansion of the productions, we apply the appropriate prime-number to the total product representing the integer. For example, if we wish to encode the value of 2, we begin with the sentential form 'S' and apply the necessary productions to result in an equivalent product. We apply production p_0 , being mapped to the prime-number 2, and multiply to a product of 2 and a sentential form of '*aSa*'. Finally, we apply production p_2 , being mapped to 1, and multiply to a product of 2 and a sentential form of '*aa*'. Consequently, we can see that *bb* is representative of $3*1=3$, *aaaa* of $2*2*1=4$, *aabbaa* of $2*2*3*1=12$, ad infinitum.

3.1 Complexity

The overall complexity of a prime-based mimic function is quite low. Storage of the grammar used in a prime-based mimic function is simply based on the number of productions used and thus linear in space complexity. Implementation, discussed in greater detail below, of a prime-based mimic function may be complex in logic, but only requires four total passes of the production list per sentence. The reason for this requirement is discussed in the implementation section. Therefore, the time complexity is defined as,

$$x = 4pn ,$$

where p is the number of productions in the grammar and n is the number of

sentence to create. Since p is constant across all n sentence creations, x exhibits linear growth and thus prime-based mimic functions are linear in time complexity.

3.2 General Usage

The general process of using a prime-based mimic function as a steganography tool is shown in figure 1. In order for Alice to send a message secretly to Bob, she must employ a steganography technique; in our case she uses a prime-based mimic function. Once transmitted, the secret message is only *hidden* not *encrypted*. If the steganography technique is known then the message is vulnerable to exposure. As usual in secret message exchange, Alice encrypts her plain-text message with message with an encryption cipher in preparation for transmission. Before transmission, the cipher-text is segmented into units of identical size. The unit size is dependent upon the mapping algorithm described above and thus dependent upon the largest prime number used by the prime-based mimic function and the largest integer to encode. Sequentially, each cipher-text unit will be processed by the prime-based mimic function into the hidden-text. This hidden-text is then transmitted to Bob. If the language of strings defined by the grammar of the prime-based mimic function is complex enough, then Eve considers the hidden-text uninteresting and ignores it. Bob then processes the hidden-text into cipher-text using the same prime-based mimic function and then decrypts the cipher-text.

1. Alice constructs a message to send to Bob. Initially, this message exists as plaintext and needs to be encrypted in order maintain secrecy of the message content.

2. Alice uses a strong encryption cipher to encrypt the plaintext message to a ciphertext message. Although unreadable, the current message state is recognizable as ciphertext.

3. Before hiding, the ciphertext is segmented into identically sized units. These identical units are factored into their prime components and sent to the prime-based mimic function for processing.

4. Alices' prime-based mimic function processes the prime components, converting the set into the hiddentext.

5. The hiddentext, being a string in the language accepted by the prime-based mimic function, becomes unrecognizable as ciphertext.

6. Eve, the snooping party, intercepts the hiddentext and ignores it, considering the text uninteresting.

7. Bob receives the hiddentext and begins processing it to obtain the ciphertext.

8. Bob's prime-based mimic function will then process the hiddentext into prime components, find their product.

9. The total transmission of each ciphertext unit is recombined into the original ciphertext message.

10. Bob uses the appropriate cipher to decrypt the ciphertext obtaining the plaintext.

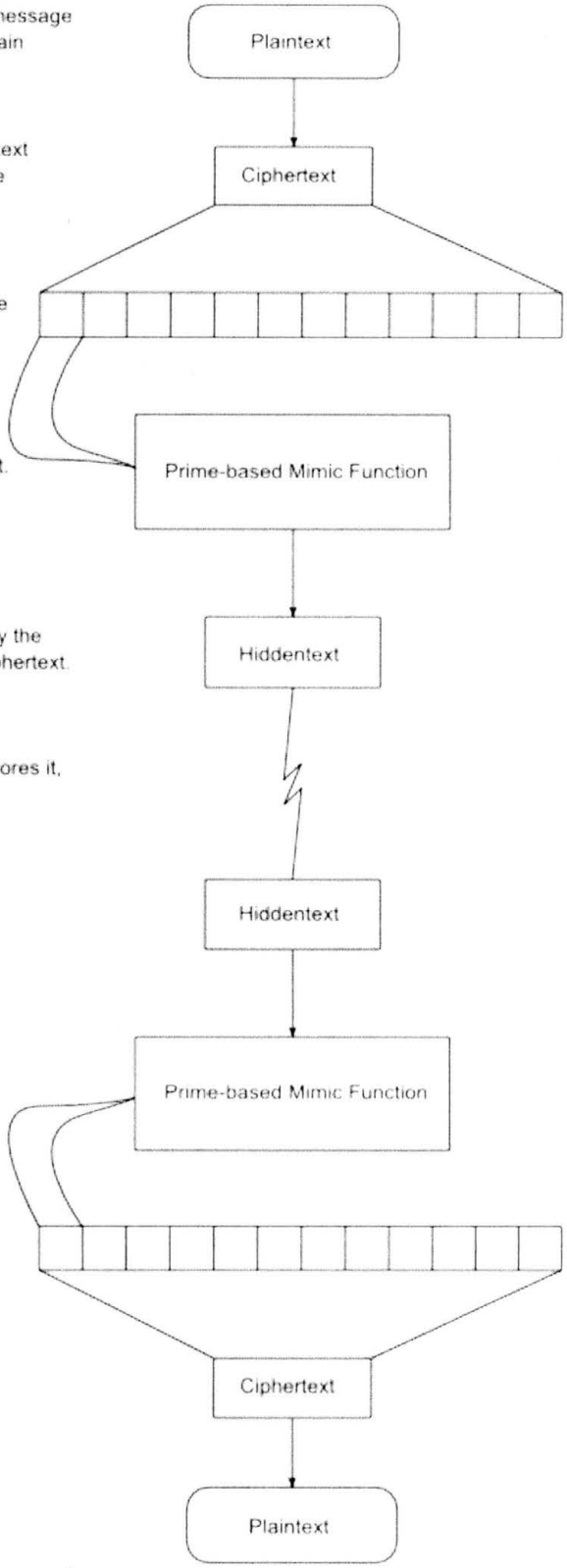

Figure 1: General usage

3.3 Mapping Algorithm

Almost immediately, the problem of sparseness is apparent. Prime numbers not related in ξ cannot be components of the integer encoded in a sentence. For instance, any prime number greater than three cannot be used. Thus, in the example given above, integers such as seven and its multiples cannot be represented. In order to overcome this problem, we define a mapping algorithm to remove the sparseness.

First, we must define a range of integers to map. The format of our input tends to dictate the maximum size of our integers. For instance, when using digital data, a power of two such as 256, is appropriate. Second any mapping algorithm can used that meets a few criteria: the mapping must be one-to-one, it must be invertible, and each unique integer must have a prime factorization containing the primes found in the production to prime-number mapping, ξ . This mapping algorithm allows the full range of input data to be mapped to an integer that can be encoded by G . An example mapping that fits with our example grammar from above is,

$$
\begin{aligned}
&0 \rightarrow 1 : \{\} \\
&1 \rightarrow 2 : \{2\} \\
&2 \rightarrow 3 : \{3\} \\
&3 \rightarrow 4 : \{2,2\} \\
&4 \rightarrow 6 : \{2,3\} \\
&5 \rightarrow 8 : \{2,2,2\} \\
&6 \rightarrow 9 : \{3,3\} \\
&7 \rightarrow 12 : \{2,2,3\} \\
&8 \rightarrow 16 : \{2,2,2,2\}
\end{aligned}
$$

The example mapping allows a 3-bit integer, including the integers five and seven, to be encoded using only the prime factors of two and three. This type of mapping algorithm,

albeit primitive, can be expanded to include any size of input integer. The bit-depth of the maximum integer accepted by the algorithm as input is, from here on, defined as the *block size* of the mimic function. During general usage, the block size is the size of the cipher-text segments.

There are, however, limitations to the mapping algorithm. Since the mapped values are typically stored in CPU registers or standard language integer variables, they are limited to the maximum sizes of these containers. For instance, the maximum size of an unsigned integer on a standard desktop machine is limited to 2^{32}. If the mapped values exceed this limitation, the container typically wraps and begins at zero which invalidates the uniqueness constraint on the mapping algorithm. We can determine the maximum integer required for a mapping by calculating,

$$y=\prod_{i=1}^{n} p_i^{\log_n(x)} \quad ,$$

where n is the cardinality of the unique set of prime numbers used in the prime-based mimic function and x is the largest integer to be encoded. As an example, let our unique set of prime numbers be, $\{2,3,5,7\}$ and our largest integer to be encoded one 8-bit byte, or 256. This evaluates to,

$$y=2^4 3^4 5^4 7^4=1944810000 \quad .$$

In this case, we would require a 32-bit container to store each mapped value. When the largest integer to be encoded is increased to two 8-bit bytes, or 65536, the evaluation is,

$$y=2^8 3^8 5^8 7^8=1125899906842624 \quad .$$

Obviously, this quickly overflows a 32-bit container but would fit nicely in a 64-bit container. Other implementations, such as binary coded decimals, or arbitrary precision computation are able to overcome these limitations since the mapping algorithm only requires the use of the multiplication operation.

3.4 Secret-to-Cover Ratio

As described previously, the secret-to-cover ratio of a steganography technique is the ratio of the secret message size to the cover message size. This ratio is a simple measurement of a steganography technique's efficiency when transmitting a secret message. For prime-based mimic functions, the lower bounds of the secret-to-cover ratio can be defined. However, since the construction of the grammar and mapping algorithm used in a prime-based mimic function greatly influences the secret-to-cover ratio, an exact ratio must be determined individually.

The lower-bound of the secret-to-cover ratio is defined as requiring one symbol in the cover message for each prime factor of the mapped value obtained from the mapping algorithm. Since a production in the grammar must have a right-hand side variable and eventually terminate with at least one symbol, a single prime factor must be expressed by at least one symbol. Therefore, the number of mapped value prime factors determine the lower-bound of the secret-to-cover ratio.

With the strictest definition of a prime-based mimic function, the upper-

bound of the secret-to-cover ratio cannot be defined since the number of possible sentences is infinite. This lack of an upper-bound is caused by the traversal of productions having a prime-cost of 1 arranged in a loop. However, an implementation would limit the traversal of these productions and reduce the upper-bound to a finite, yet still possibly large, value. In fact, in our experiment the secret-to-cover ratio exhibited was between 0.028 and 0.069, a value greater than most of the steganography techniques described in the literature survey.

3.5 Robustness

As described previously, robustness is the ability of a steganography technique to resist modification to the cover message. Fortunately, prime-based mimic functions exhibit a competitive level of resistance to modification of the generated cover message. However, this resistance is dependent upon the grammar and mapping algorithm used for the prime-based mimic function.

A prime-based mimic function resists modification to the generated cover message if the changed symbols are part of productions with the same prime-cost. If, during parsing, the changed symbol causes the decoder to choose a path with the same prime-cost, then the change has been successfully resisted. For instance, if we assume a grammar includes the two productions, $K \rightarrow y, K \rightarrow x$, each with an equal prime-cost, then the replacement of x with y or y with x has no effect on the final product.

When the modification is bit manipulation instead of symbol manipulation, a different quality of robustness is present. The two types of bit manipulation are when a bit in the data stream has been flipped and when a bit is missing. If a bit within the data stream has been flipped, the entire sentence is invalid and must be retransmitted. If a bit is missing from the data stream, the end of the current expression cannot be properly determined invalidating the entire data stream. Error correction bits and symbol escaping can be used to further resist modifications of this kind. However, this topic is outside the scope of this thesis.

4. Experiments

In this section we describe our implementation details and experiment methodology. We also describe the techniques used in analyzing the results of our experiments.

4.1 Implementation

Each prime-based mimic function requires a language definition to serve as the cover for our secret data. In our case, we implement our mimic function to produce strings within a basic arithmetic expression language. A context-free arithmetic expression language can be defined unambiguously and easily meets the prime-numbering criteria discussed above; thus, the language definition is quite suitable for our implementation purposes. We define the basic arithmetic language as:

$$\text{Let } G=(V,T,E,P) \text{ such that,}$$

$$V=\{E,T,F,G,K\},$$

$$T=\{x,y,+,-,*,/,\wedge,\$\},$$

and with productions,
$E \rightarrow T$
$T \rightarrow F$
$F \rightarrow G$
$G \rightarrow K$
$K \rightarrow x|y$
$E \rightarrow E+T$
$E \rightarrow E-T$
$T \rightarrow T*F$
$T \rightarrow T/F$
$F \rightarrow F \wedge G$
$F \rightarrow F \$ G$
$G \rightarrow (E).$

The format of the strings contained within this language is similar to many of the expressions found in a typical college algebra textbook. Two aesthetic changes have been made to the syntax to facilitate consistency for the implementation. If a single character is used to denote an operation, tokenization and parsing can be greatly simplified, especially if every operation is a single character. The exponential operation has been denoted with a caret character and the logarithm operation has been denoted with a dollar sign character. A few examples are:

$$(x+y)^x \text{ is denoted as } (x+y) \wedge x$$

$$\log_x(\frac{y}{x}) \text{is denoted as } x \$ (y/x)$$

We now define the prime-number to production mapping function as:

$$\delta = \{\{P_0,1\},\{P_1,1\},\{P_2,1\},\{P_3,1\},\{P_4,1\},\{P_5,2\},$$
$$\{P_6,3\},\{P_7,5\},\{P_8,7\},\{P_9,2\},\{P_{10},3\},\{P_{11},1\}\}$$

Fortunately, the choice of prime-number mapping for the basic arithmetic expression language allows for a simplified parsing mechanism. In the event that an

operational token is encountered during decoding, the respective prime number can be added to the list of factors. This choice is deliberate to simplify the implementation and and must be avoided in a real-world scenario. Using the previous examples, we can see that

$$(x+y)\hat{}x=2*2=4$$

$$x\$(y/x)=3*7=21 \quad .$$

As an example, we show the production applications along with the current running product of the expansion of $(x+y)\hat{}x$. We begin with the sentential form 'E',

Table 1: Example prime-based mimic function expansion

Production	Sentential Form	Mapped Prime-number	Running Product
$E \rightarrow T$	T	1	1
$T \rightarrow F \hat{} G$	$F\hat{}G$	2	2
$F \rightarrow G$	$G\hat{}G$	1	2
$G \rightarrow (E)$	$(E)\hat{}G$	1	2
$E \rightarrow E+T$	$(E+T)\hat{}G$	2	4
$E \rightarrow T$	$(T+T)\hat{}G$	1	4
$T \rightarrow F$	$(F+T)\hat{}G$	1	4
$F \rightarrow G$	$(G+T)\hat{}G$	1	4
$G \rightarrow K$	$(K+T)\hat{}G$	1	4
$K \rightarrow x$	$(x+T)\hat{}G$	1	4
$T \rightarrow F$	$(x+F)\hat{}G$	1	4
$F \rightarrow G$	$(x+G)\hat{}G$	1	4
$G \rightarrow K$	$(x+K)\hat{}G$	1	4
$K \rightarrow y$	$(x+y)\hat{}G$	1	4
$G \rightarrow K$	$(x+y)\hat{}K$	1	4
$K \rightarrow x$	$(x+y)\hat{}x$	1	4

We implement prime-based mimic function encoder and decoder in C using the GLib library [15]. The GLib library was chosen for its portable and mature

implementation of hash tables, linked lists, and string utilities. Specifically, the linked list implementation in the GLib library is helpful since a great deal of list iteration is required in the encoder portion of the mimic function implementation. Additionally, the hash table implementation is used in the transmission mapping implementation.

The mimic function implementation, named the `mimic-coder`, accepts two command-line option flags for encoding and decoding with a required filename argument. If the encoding flag has been set, then the file is read a single byte, assuming an 8-bit byte, at a time. Each byte is encoded into an arithmetic expression and printed to standard output. If the decoding flag has been set, then the file is read a line at a time , assuming newline terminated lines. Each line is decoded into a value and printed to standard output. Any program or operating system errors are printed to standard error and cause termination of the program.

In order to encode a byte of data in an arithmetic expression, four sets of productions must be found and added to the final set of productions to apply to the initial sentential form of 'E'. These four sets of productions are: the required productions, the pre-bridge productions, the bridge productions, and the post-bridge productions. The required productions are those productions that are required to encode every prime-factor for the current byte's transmission value. The pre-bridge productions are those productions required to derive from the initial sentential form to the first production in the required productions set. Occasionally, the pre-bridge production set is empty since

the first production in the required production set matches the initial sentential form. The bridge productions are those productions required to derive from the one of the right-hand side symbols of a required production to a following required production. These productions connect the derivation of each required production. Any remaining right-hand side symbols which have not been matched to other required productions are matched to productions leading to terminals.

First, the byte's integer value is converted to the transmission value and its respective prime-factor set. The prime-factor set is then iterated and a random production matching each prime-factor is added to required productions set. A random production that matches a specific prime-factor is helpful in strengthening the resulting sentential form. After each prime-factor has been matched to a required production, the pre-bridge productions are found connecting the derivation of the initial sentential form to the first required production. Next, the bridge productions are found, followed by the post-bridge productions. Finally, with the final set of productions, the sentential form is derived and printed to standard out.

4.2 Methodology

In our experiment, the SRI Language Modeling toolkit, SRILM toolkit, was used to analyze five sets of data encoded by the `mimic-coder` program: uncompressed, compressed, encrypted, random, and zero. Each data set was further analyzed using a 4-bit, 8-bit, and 16-bit block size for the intermediate value mapping

algorithm. A 4-bit block size effectively doubled the number of arithmetic expressions produced while the 16-bit block size halved the number of arithmetics expressions produced. Thus, a 100 byte file produces 200, 100, and 50 arithmetic expressions for 4-bit, 8-bit, and 16-bit block sizes respectively. The following is a listing of all experiments performed:

- Compressed ML with a 4-bit Block Mapping
- Compressed ML with a 8-bit Block Mapping
- Compressed ML with a 16-bit Block Mapping
- Encrypted ML with a 4-bit Block Mapping
- Encrypted ML with a 8-bit Block Mapping
- Encrypted ML with a 16-bit Block Mapping
- Uncompressed ML with a 4-bit Block Mapping
- Uncompressed ML with a 8-bit Block Mapping
- Uncompressed ML with a 16-bit Block Mapping
- /dev/urandom Data with a 4-bit Block Mapping
- /dev/urandom Data with a 8-bit Block Mapping
- /dev/urandom Data with a 16-bit Block Mapping
- One Valued Byte Data with a 4-bit Block Mapping
- One Valued Byte Data with a 8-bit Block Mapping
- One Valued Byte Data with a 16-bit Block Mapping
- /dev/zero Data with a 4-bit Block Mapping
- /dev/zero Data with a 8-bit Block Mapping
- /dev/zero Data with a 16-bit Block Mapping

Two additional experiments were performed to compare the results of the above experiments with valid and invalid arithmetic expressions. In order to generate the valid arithmetic expressions, the derivation tree of the basic arithmetic language was walked using a depth-limited search. By choosing a depth of 20, we were able to generate 77,050 arithmetic expressions to be analyzed by SRILM. In order to generate the invalid arithmetic expressions, a set of sentences with random lengths no greater than 10 symbols were generated. The symbols chosen were those existing in the language. If symbols

outside the language were used, the language model implemented by SRILM would automatically give an extremely high perplexity.

The output of the SRILM package was then processed with a trivial Perl script to extract the statistical results of the analysis. Usage of the SRILM requires the creation of a language model trained with a corpus which is then applied to the test data. The training data used was a set of one-hundred arithmetic expressions from a college algebra textbook. The default language models and parameters present for the SRILM package were found to be suitable. The general input chosen for the `mimic-coder` program is page 270 of The Memoirs and Letters Benjamin Franklin (ML). In this usage, the ML serves as the secret message while the arithmetic expressions generated by the program serve as the cover. Notably, this specific page contains Benjamin Franklin's quote, "They who can give up essential liberty to obtain a little temporary safety, deserve neither liberty nor safety."

The total size of the ML, representative of the uncompressed data set, is 12,404 bytes. Since the ML is encoded as standard ASCII text, the range of values is limited to the printable characters. For the compressed data set, the ML was compressed using the default configuration of the `gzip` compression application and resulted in a data file 5,149 bytes in size. The encrypted data set, created by processing the ML with `openssl enc -aes-256-cbc -salt`, produced a file larger than the original text at 12,432 bytes. For comparison, 12,404 bytes of random data, 12,404 bytes of one

valued bytes and 12,404 bytes of zeros were encoded separately to be analyzed. The random data bytes were dumped from the `/dev/urandom` device on a server running a standard Linux kernel version 2.6.27. The zero data bytes were dumped from the `/dev/zero` device on the same machine.

After processing the data from the experiments into a usable form, the analysis compares the geometric and average perplexities for each respective experiment. Perplexity data points which are closer to the origin of the graph show less surprise on the part of the language model and thus an expected sentence. The comparison of perplexities has been used by Meng [19] and Taskiran [20] to automatically and accurately identify generated cover-text from normal text. Both Meng and Taskiran use the SRILM Toolkit to analyze generated cover-text and normal text. Finally, the data sizes before and after encoding were compared across experiments to show relationships between the block sizes of the mapping algorithm and how well each type of data encodes with respect to bit-rate.

5. Results

In this section we describe the results obtained from our experiments. Each experiment is shown on a single page with an explanation of the results for clarity. For each graph, a general description is provided along with a notice regarding outlying data points, clustered data points, and maximum perplexities.

5.1 Compressed ML with a 4-bit Block Mapping

Figure 2 shows the results of an experiment using a compressed data file of the ML text and a 4-bit block in the mapping algorithm. As defined earlier, perplexity is the relative surprise of the language model when analyzing a sentence. We see a tight clustering of data points within (30, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 53 geometric perplexity or an average perplexity of 243. Four outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

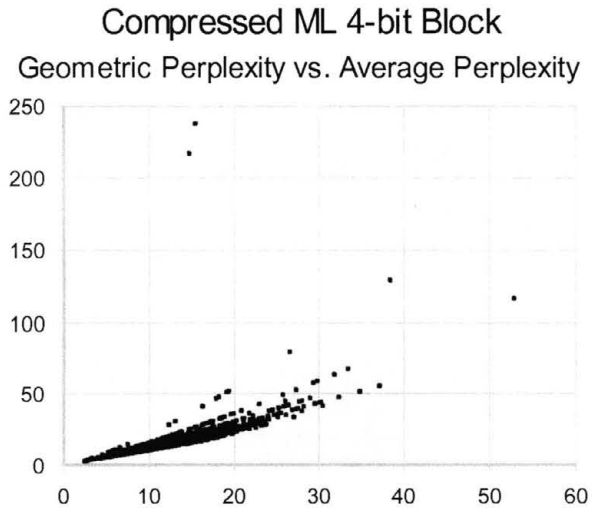

Figure 2: Compressed ML 4-bit Block

5.2 Compressed ML with a 8-bit Block Mapping

Figure 3 shows the results of an experiment using a compressed data file of the ML text and a 8-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 37 geometric perplexity or an average perplexity of 243. Three outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

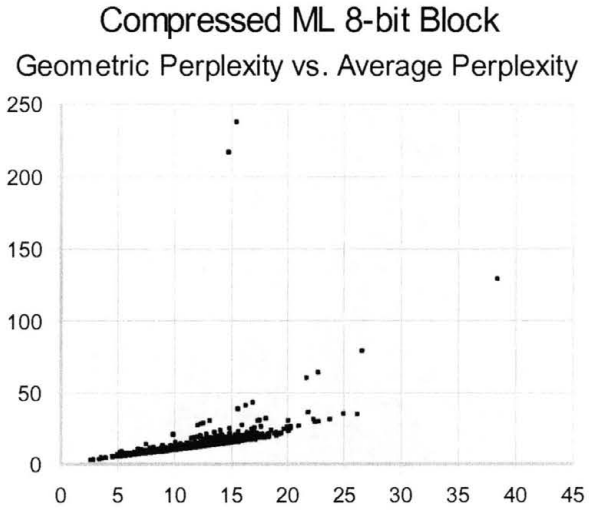

Figure 3: Compressed ML 8-bit Block

5.3 Compressed ML with a 16-bit Block Mapping

Figure 4 shows the results of an experiment using a compressed data file of the ML text and a 16-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 52 geometric perplexity or an average perplexity of 243. Four outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

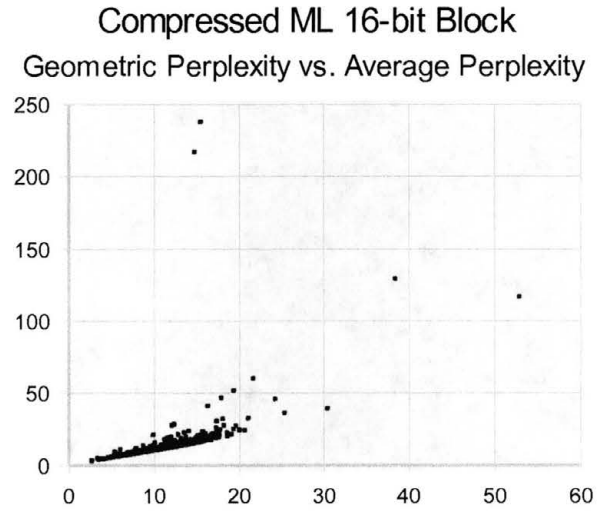

Figure 4: Compressed ML 16-bit Block

5.4 Encrypted ML with a 4-bit Block Mapping

Figure 5 shows the results of an experiment using a encrypted data file of the ML text and a 4-bit block in the mapping algorithm. We see a tight clustering of data points within (30, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 37 geometric perplexity or an average perplexity of 243. Three outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

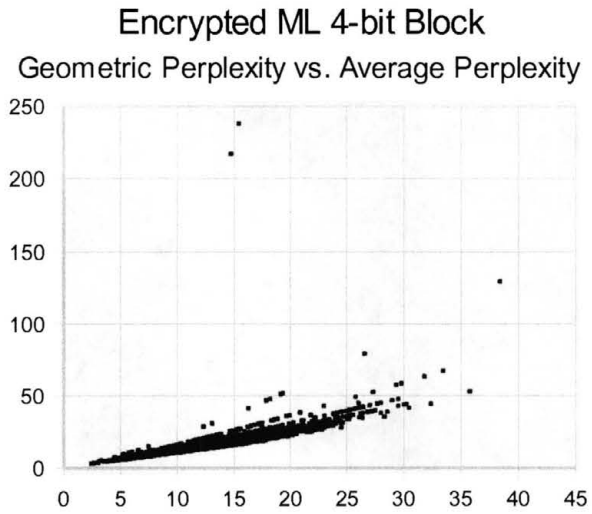

Figure 5: Encrypted ML 4-bit Block

5.5 Encrypted ML with a 8-bit Block Mapping

Figure 6 shows the results of an experiment using a encrypted data file of the ML text and a 8-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 37 geometric perplexity or an average perplexity of 243. Three outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

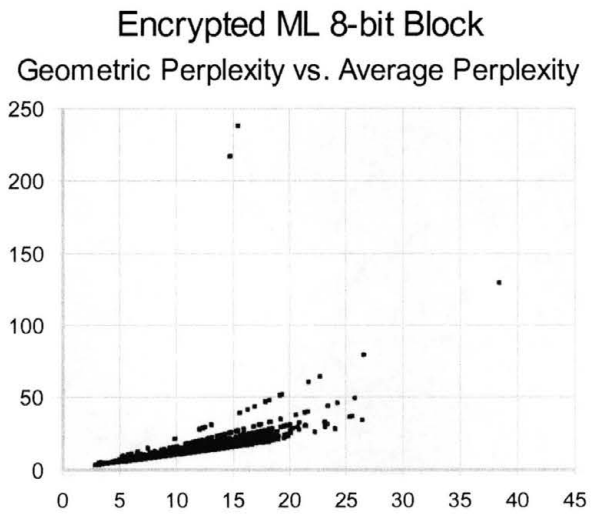

Figure 6: Encrypted ML 8-bit Block

5.6 Encrypted ML with a 16-bit Block Mapping

Figure 7 shows the results of an experiment using a encrypted data file of the ML text and a 16-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 37 geometric perplexity or an average perplexity of 243. Three outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

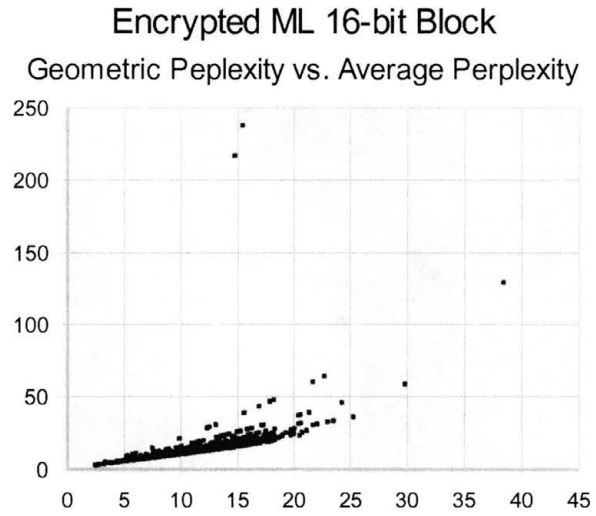

Figure 7: Encrypted ML 16-bit Block

5.7 Uncompressed ML with a 4-bit Block Mapping

Figure 8 shows the results of an experiment using a plain-text data file of the ML text and a 4-bit block in the mapping algorithm. We see a tight clustering of data points within (30, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 53 geometric perplexity or an average perplexity of 243. Three outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

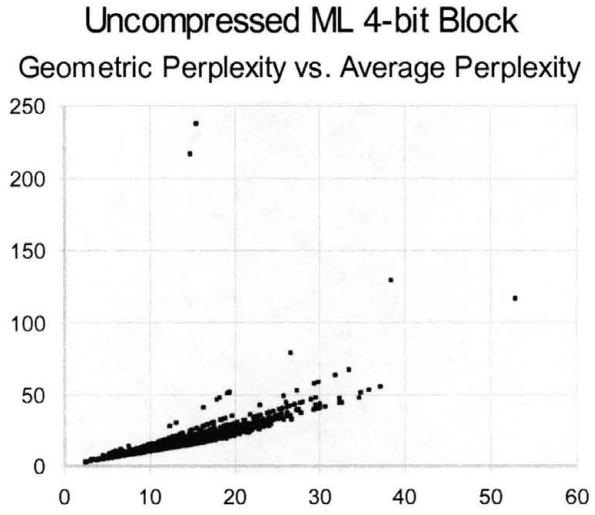

Figure 8: Uncompressed ML 4-bit Block

5.8 Uncompressed ML with a 8-bit Block Mapping

Figure 9 shows the results of an experiment using a plain-text data file of the ML text and a 8-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 25) to the origin which shows the language model accepting the sentences with very little surprise. No outliers exist and no data points exceed a 25 geometric perplexity or an average perplexity of 40.

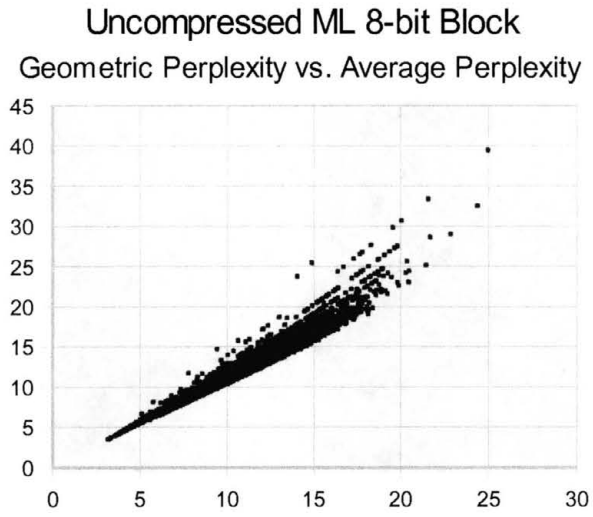

Figure 9: Uncompressed ML 8-bit Block

5.9 Uncompressed ML with a 16-bit Block Mapping

Figure 10 shows the results of an experiment using a plain-text data file of the ML text and a 16-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 25) to the origin which shows the language model accepting the sentences with very little surprise. No outliers exist and no data points exceed a 22 geometric perplexity or an average perplexity of 33.

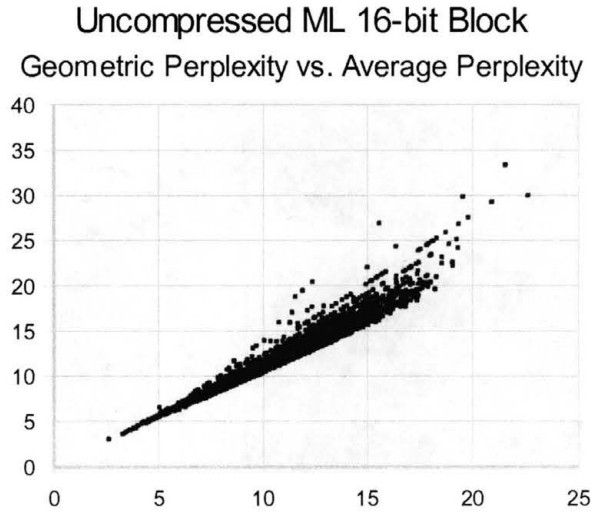

Figure 10: Uncompressed ML 16-bit Block

5.10 /dev/urandom Data with a 4-bit Block Mapping

Figure 11 shows the results of an experiment using a data file of 12,404 bytes of random data and a 4-bit block in the mapping algorithm. We see a tight clustering of data points within (30, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 53 geometric perplexity or an average perplexity of 243. Three outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

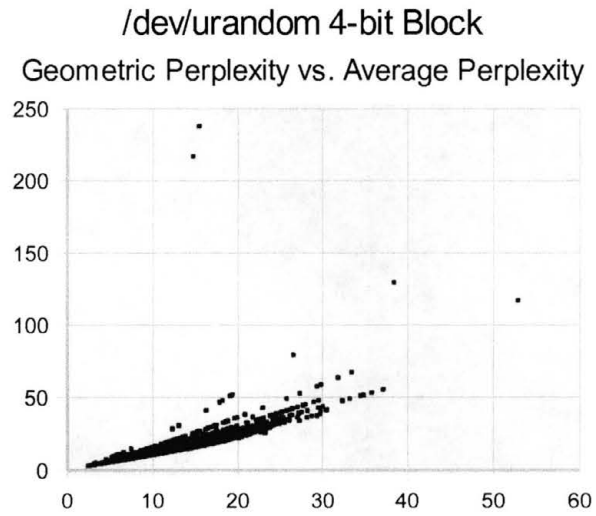

Figure 11: /dev/urandom 4-bit Block

5.11 /dev/urandom Data with a 8-bit Block Mapping

Figure 12 shows the results of an experiment using a data file of 12,404 bytes of random data and a 8-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 53 geometric perplexity or an average perplexity of 243. Four outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

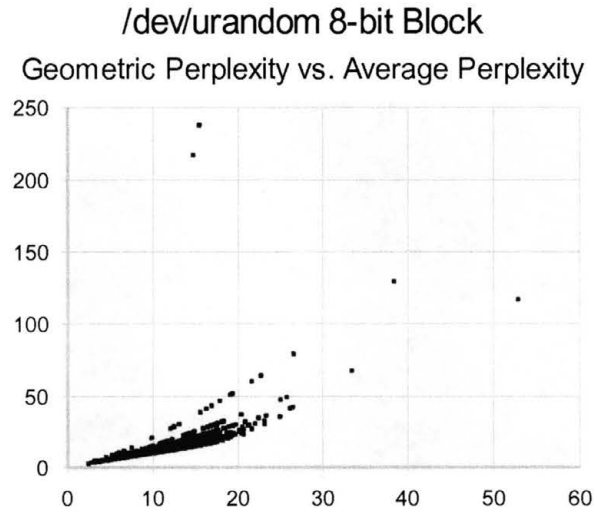

Figure 12: /dev/urandom 8-bit Block

5.12 /dev/urandom Data with a 16-bit Block Mapping

Figure 13 shows the results of an experiment using a data file of 12,404 bytes of random data and a 16-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 50) to the origin which shows the language model accepting the sentences with little surprise. No data points exceed a 33 geometric perplexity or an average perplexity of 243. Four outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

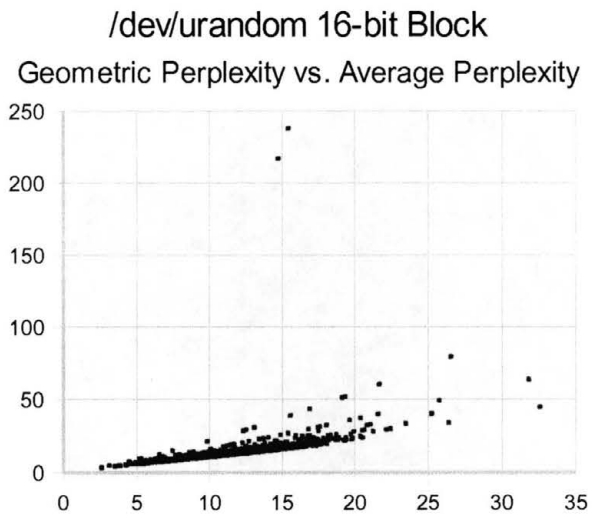

Figure 13: /dev/urandom 16-bit Block

5.13 One Valued Byte Data with a 4-bit Block Mapping

Figure 14 shows the results of an experiment using a data file of 12,404 bytes all with a value of one and a 16-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 50) to the origin which shows the language model accepting the sentences with little surprise. However, the clustering is flatter than the previous experiments and shows lower boundary of the perplexities. No data points exceed a 37 geometric perplexity or an average perplexity of 243. Three outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

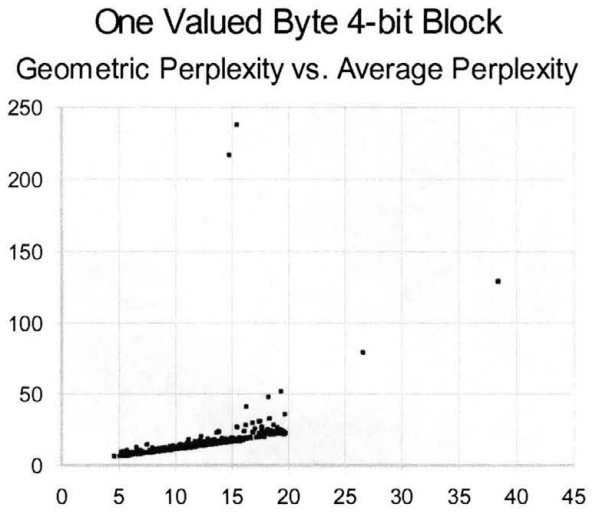

Figure 14: One Valued Byte 4-bit Block

5.14 One Valued Byte Data with a 8-bit Block Mapping

Figure 15 shows the results of an experiment using a data file of 12,404 bytes all with a value of one and a 16-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 40) to the origin which shows the language model accepting the sentences with little surprise. However, the clustering is flatter than the previous experiments and shows lower boundary of the perplexities. No data points exceed a 37 geometric perplexity or an average perplexity of 130. One outlying data point is present which represents mapped values of 1 and show both higher geometric and average perplexities.

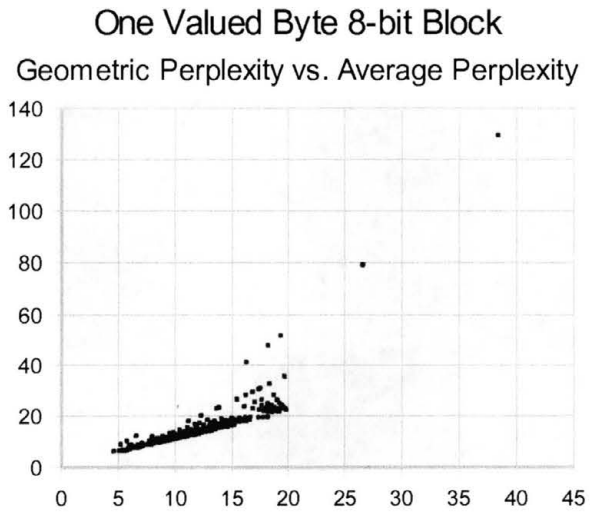

Figure 15: One Valued Byte 8-bit Block

5.15 One Valued Byte Data with a 16-bit Block Mapping

Figure 16 shows the results of an experiment using a data file of 12,404 bytes all with a value of one and a 16-bit block in the mapping algorithm. We see a tight clustering of data points within (20, 30) to the origin which shows the language model accepting the sentences with little surprise. However, the clustering is flatter than the previous experiments and shows lower boundary of the perplexities. No data points exceed a 37 geometric perplexity or an average perplexity of 130. One outlying data point is present which represents mapped values of 1 and show both higher geometric and average perplexities.

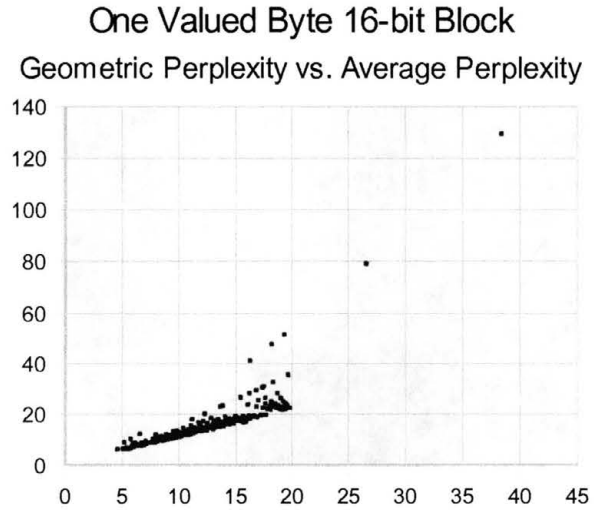

Figure 16: One Valued Byte 16-bit Block

5.16 /dev/zero Data with a 4-bit Block Mapping

Figure 17 shows the results of an experiment using a data file of 12,404 bytes all with a value of zero and a 4-bit block in the mapping algorithm. We see a sparse and roughly linear layout of data points lying between (16, 25) and the origin which shows the language model accepting the sentences with very little surprise. No data points exceed a 17 geometric perplexity or an average perplexity of 243. Two outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

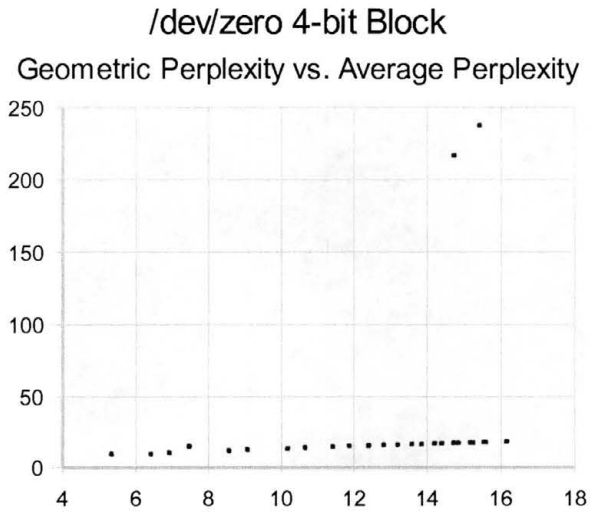

Figure 17: /dev/zero 4-bit Block

5.17 /dev/zero Data with a 8-bit Block Mapping

Figure 18 shows the results of an experiment using a data file of 12,404 bytes all with a value of zero and a 8-bit block in the mapping algorithm. We see a sparse and roughly linear layout of data points lying between (16, 25) and the origin which show the language model accepting the sentences with very little surprise. No data points exceed a 16 geometric perplexity or an average perplexity of 243. Two outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

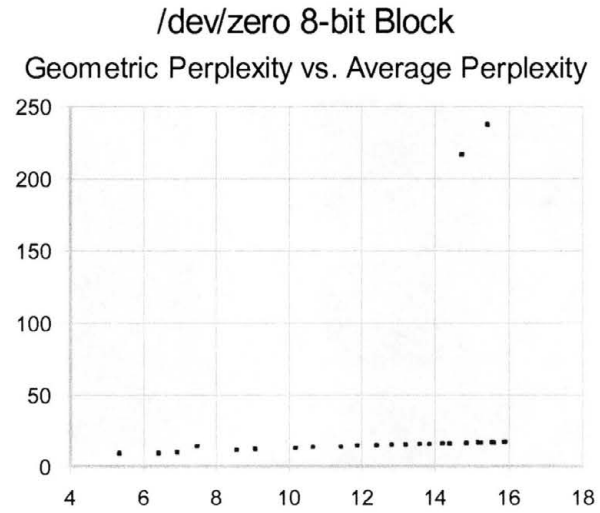

Figure 18: /dev/zero 8-bit Block

5.18 /dev/zero Data with a 16-bit Block Mapping

Figure 19 shows the results of an experiment using a data file of 12,404 bytes all with a value of zero and a 16-bit block in the mapping algorithm. We see a sparse and roughly linear layout of data points lying between (16, 25) and the origin which shows the language model accepting the sentences with very little surprise. No data points exceed a 16 geometric perplexity or an average perplexity of 243. Two outliers are present which represent mapped values of 1 and show both higher geometric and average perplexities.

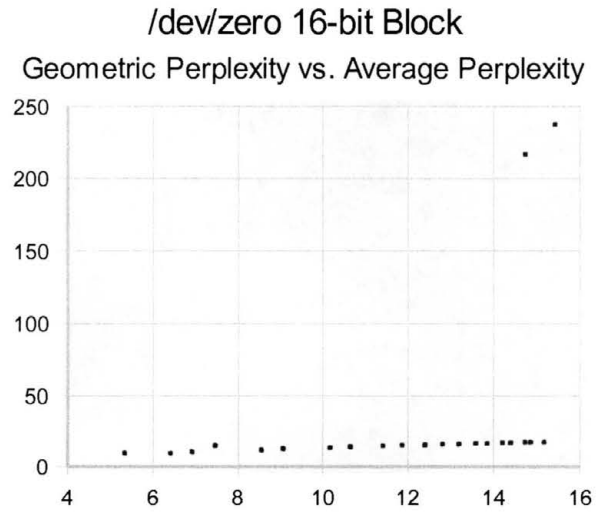

Figure 19: /dev/zero 16-bit Block

5.19 Experiment Summary

The table below summarizes the results of the experiments by showing the starting range, ending range, and brief notes about distribution of data points. The two additional experiments using valid and invalid expression, although not graphed, have been listed in the table. Descriptions within the 'Notes' column describe the overall distribution of the data set. Outliers, implies that the data is clustered with a few points located far from the cluster. Thin, implies that the data are not only clustered, but clustered tightly in a thin band. Sparse, implies that the data is not clustered. Uniform, implies that the data are spread throughout the range of the data.

Table 2: Experiment Summary

Experiment Name	Starting Range	Ending Range	Notes
Comp. ML, 4-bit	(0,0)	(30, 50)	Outliers
Comp. ML, 8-bit	(0,0)	(20, 50)	Outliers
Comp. ML, 16-bit	(0,0)	(20, 50)	Outliers
Enc. ML, 4-bit	(0,0)	(30, 50)	Outliers
Enc. ML, 8-bit	(0,0)	(20, 50)	Outliers
Enc. ML, 16-bit	(0,0)	(20, 50)	Outliers
Unc. ML, 4-bit	(0,0)	(30, 50)	Outliers
Unc. ML, 8-bit	(0,0)	(20, 25)	No Outliers
Unc. ML, 16-bit	(0,0)	(20, 25)	No Outliers
Random, 4-bit	(0,0)	(30, 50)	Outliers
Random, 8-bit	(0,0)	(20, 50)	Outliers
Random, 16-bit	(0,0)	(20, 50)	Outliers
One Valued, 4-bit	(0,0)	(20, 50)	Outliers, Thin
One Valued, 8-bit	(0,0)	(20, 40)	Outliers, Thin
One Valued, 16-bit	(0,0)	(20, 30)	Outliers, Thin
Zero Valued, 4-bit	(0,0)	(16, 25)	Outliers, Sparse
Zero Valued, 8-bit	(0,0)	(16, 25)	Outliers, Sparse
Zero Valued, 16-bit	(0,0)	(16, 25)	Outliers, Sparse
Valid Expressions	(0,0)	(72, 118)	Uniform
Invalid Expressions	(0,0)	Undefined	Sparse

5.20 Average Bit-rate Comparison

Figure 21 shows a column graph comparing the secret-to-cover ration for the encrypted, random, and uncompressed experiments. The compressed, zero and one-valued byte experiments were omitted since they transmitted no data.

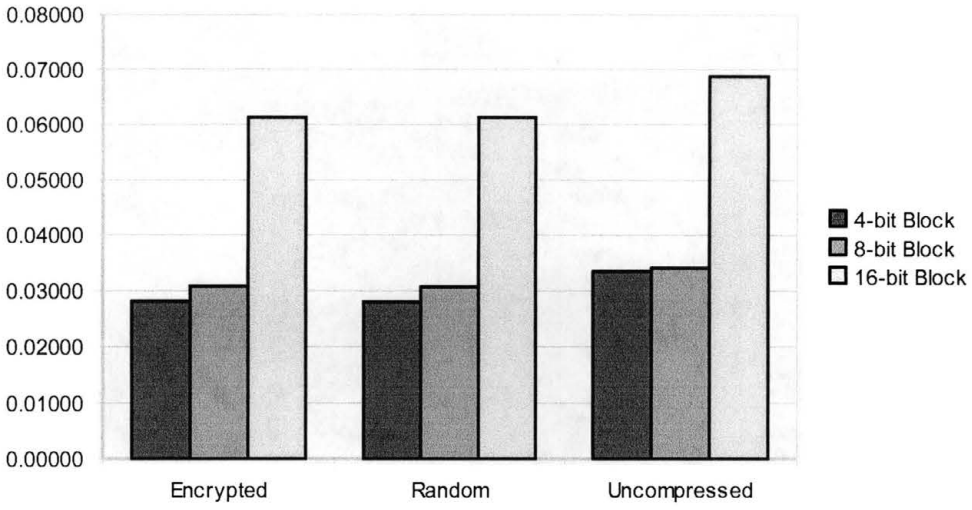

Figure 20: Secret-to-cover Ratio Comparison

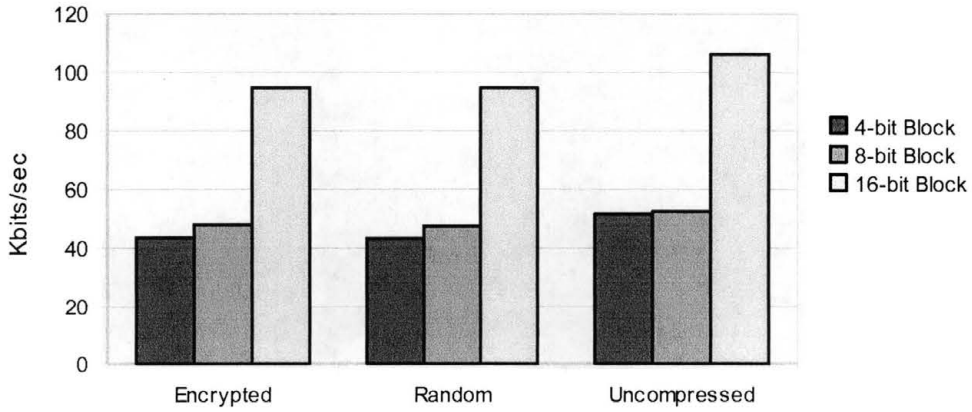

Figure 21: Average bit-rate comparison

Figure 20 shows a column graph comparing the average bit-rate across a T1 telecommunications line for each experiment type and block size. The graph shows the 4-bit and 8-bit block sizes having similar bit-rates while the 16-bit block have a doubled bit-rate. Additionally, the ratios between the bit-rate of each block size, with the

exception of the uncompressed results, are nearly identical at a 9% increase between 4-bit and 8-bit and 50% between 8-bit and 16-bit.

6. Evaluation and Conclusions

In this section we evaluate and discuss the results of our experiments. Primarily, we discuss the clustering and outlying data points relating to perplexity and the comparison of bit-rates between experiments and against other steganography techniques. Our criteria for success are how well the tested output of prime-based mimic functions compare to other known perplexities and how well the bit-rate compares to other steganography techniques.

One of the first properties of the data gathered is the consistent clustering exhibited. The majority of perplexity measurements lie below a geometric perplexity of 30 and an average perplexity of 50. With the exception of the `/dev/zero` experiments, the perplexity measurements are largely identical. The outlying measurements, and the entire data set of the `/dev/zero` experiments, were expanded from the value of zero which mapped to the transmitted value of one and thus a simple path through the grammar. Due to the definition of the arithmetic grammar, a simple path results in only a few combinations of expanded sentential forms.

When comparing the clusters of perplexity measurements to other external measurements, a prime-based mimic function performs quite well. When using a highly

trained English-based language model, Katz [16] reported perplexity measurements between 80 and 120 during a comparison of their language modeling technique and other common techniques. Another report by Brown [17], found that the Brown linguistic corpus exhibited a perplexity of 271 and later perplexities of 244 and 236 after data interpolation was applied.

While the English-based language models exhibited higher perplexities with test data, they aren't completely applicable for comparison to the results of our experiments. The analysis of the generated valid and invalid arithmetic expressions shows that all of the sentences generated in the ML experiments, with the exception of the few outliers, exhibit a similar range of perplexities. Additionally, the experiments with random data, one-valued data, and zero-valued data also exhibited a similar range of perplexities to that of the generated valid arithmetic expressions.

This shows that the expressions generated by our prime-based mimic function are statistically close to actual arithmetic grammar. We can see from figures 8, 9, and 10, that the uncompressed transmission exhibited the best clustering of perplexity measurements. Since the uncompressed data are in ASCII format and contained no control characters, the chance of encountering a zero valued byte is nil.

Of the many steganography techniques, only a few had a high secret-to-cover ratio. Particularly, the Least Significant Bit substitution technique yielded a secret-to-

cover ratio of 0.43 or a 0.633 Mbit/sec bit-rate which greatly outperformed all other techniques by several orders of magnitude. This lack of throughput in the other techniques can be explained by the limiting of a single bit per decision, choice, or segment. In some cases, the segment was the entire cover medium, which resulted in an extremely low bit-rate. Fortunately for prime-based mimic functions, the bit-rate performance increases as the language complexity increases, as the number of expressible prime numbers increases, and as the block size of the mapping algorithm increases. We can see this behavior in figure 20; the `/dev/zero` experiment was ignored since no actual information was transmitted.

Of the four remaining experiments analyzed for bit-rate, the highest performing experiment was the 'Uncompressed ML 16-bit Block', shown in figure 4 with a secret-to-cover ratio of 0.069 or a bit-rate of 106.26 Kbits/sec. As a reminder, these secret-to-cover ratios and bit-rate calculations are based on a T-1 telecommunications line and are determined by the discussion in section 3.4. A remarkable result of varying the block size is shown in the increasing bit-rate when moving to a large block size. Although there were not enough block size experiments performed to infer a formula, a near exponential growth can be extrapolated. With the compressed, encrypted, and random experiments, the increase between a 4-bit and 8-bit block size was nearly identical with a 9% growth in bit-rate. Since the compression and encryption tend to increase the entropy of a signal, it would be expected that these experiments would behave similarly to each other. Meanwhile, the bit-rate in the uncompressed experiment

grew only 2% from the 4-bit block to the 8-bit block. When moving to a 16-bit block mapping algorithm, the bit-rate in the compressed, encrypted, and random experiments grew approximately 50% while the uncompressed experiment grew a close 49%. Obviously, using a larger block size results in a much higher bit-rate for transmission. This is explained by the increase in mappable numbers while maintaining the same number of prime factors available for encoding.

6.1 Conclusions

By implementing a prime-based mimic function, compressed, cipher-text, and plain-text can be encoded into sentences accepted by a context-free grammar while having a low perplexity and a competitive bit-rate. The primary limitations are the implementation and execution of the mapping algorithm and construction of a context-free grammar complex enough to interesting. The secondary limitations are maintaining a suitable bit-rate and avoiding larger geometric and average perplexities.

7. Recommendations

In this section we make recommendations for future work to improve the definition and implementation of prime-based mimic functions and the associated mapping algorithm. Additionally, we discuss potential high-value uses for prime-based mimic functions provided the various limitations are overcome.

Currently the mapping algorithm requires a precomputed set of numbers and factors for each value to map, or 2^n entries where n is the block size in bits. When 4, 8, and 16-bit blocks are used, the map is small and reasonable for modern desktop and even mobile hardware to store. Desktop hardware can even manage a block size of 32-bits but are incapable of using block sizes of 64-bits or higher. A function which computes a mapping of an integer, given a specific set of prime-numbers, and runs in logarithmic or constant time and space would allow higher block sizes to be explored. If a function cannot be developed, partial table generation could be helpful since only a very small percentage of the integer space is used. Also, as discussed above, binary coded decimal methods and arbitrary precision methods could be implemented to overcome the upper bounds of mapped values.

Our experiments showed that a possible exponential growth exists as a function of the block size. Greater granularity in the experiments would support this

hypothesis. Thus, testing with sequential block sizes, such as 1, 2, 3, ..., might show a definite rate of increase in the bit-rate. Coupled with a mapping function that ran in non-exponential time and space, the upper bounds of the test could be increased dramatically and explore the behavior of the bit-rate as it approaches the maximum bit-rate of the channel. One optimistic hypothesis is that the rate is in fact exponential and the bit-rate increases at the expense of complexity in the mimic function grammar. However, the seasoned hypothesis is that the bit-rate increases will asymptotically approach the maximum bit-rate of the channel.

The choice of a grammar which implements an arithmetic language was made because it is well defined, unambiguous, well-known, and has many example sentences Now that the statistical nature of prime-based mimic functions has been explored, a grammar of substantial complexity should be constructed to explore complexity questions. Suitable choices would include the grammar for a programming language, such as C or Java, and markup languages, such as HTML or XML. In the case of C and Java, given a properly constructed grammar, the resulting output should be compilable, although not necessarily executable. HTML would be an excellent transport mechanism for a covert message as it ubiquitous on the Internet.

Additional classes of grammars should also be explored. Regular languages are a trivial case since they are a simple subset of context-free grammars and quite honestly mildly uninteresting for our purposes since extensive real world usage is limited.

However, unrestricted grammars can be adapted, using the same techniques to build Turing machines which can encode and decode highly complex sentences from any language. During the research for this thesis, sample Turing machines were built to encode and decode sentences in a very simplistic unrestricted grammar. The implementation of these machines can be daunting. Fortunately, the potential to apply prime-based mimic functions to any language exists. Additionally, [11] suggests the same for mimic functions.

Another concept developed during the research for this thesis involved the automatic mapping of productions in the grammar to prime numbers. An algorithm to perform this automatic number is defined as:

1. For every left-hand side symbol appearing in the grammar, find the lowest cost path to a terminal node.
2. For each path found, each node within the path is given a prime-cost of 1.
3. The remaining productions may be given a prime-cost of 1 or given a cost equal to a prime number.

This provides an "escape path" for each left-hand side symbol while allowing prime numbers to be mapped to the remaining productions. Future work on this algorithm would ask if this approach constructs a suitable prime-based mimic function and would attempt an implementation.

REFERENCES

[1] F. Petitcolas, R.J. Anderson, and M.G. Kuhn, "Information hiding - a survey," *Proceedings of the IEEE*, vol. 87, 1999, pp. 1062-1078.

[2] Christopher D. Manning and Hinrich Schütze, *Foundations of Statistical Natural Language Processing*, Massachusetts Institute of Technology, 1999.

[3] *SRI Language Modeling Toolkit*, STAR Laboratory.

[4] S. Katzenbeisser and F. Petitcolas, *Information Hiding: techniques for steganography and digital watermarking*, Artech House, Inc., 2000.

[5] L. Chang and I. Moskowitz, "Critical analysis of security in voice hiding techniques," *Lecture notes in computer science*, 1997, pp. 203-216.

[6] D. Gruhl, A. Lu, and W. Bender, "Echo hiding," *Lecture notes in computer science*, vol. 1174, 1996, pp. 295-316.

[7] M. Guirguis and J. Valdez, "Masquerading a Wired Covert Channel into a Wireless-like Channel," *Proceedings of the 1st IEEE International Workshop on Network Security and Privacy*, 2008.

[8] S. Cabuk, C. Brodley, and C. Shields, "IP covert timing channels: design and detection," *Proceedings of the 11th ACM conference on Computer and Communications Security*, 2004, pp. 178-187.

[9] T. Holotyak, J. Fridrich, and S. Voloshynovskiy, "Blind statistical steganalysis of additive steganography using wavelet higher order statistics," *Lecture notes in computer science*, vol. 3677, 2005, p. 273.

[10] J. Fridrich, M. Goljan, and D. Soukal, "Higher-order statistical steganalysis of palette images," *Security and Watermarking of Multimedia Contents V*, vol. 5020, 2003, pp. 178-190.

[11] Peter Wayner, "Mimic Functions," *Cryptologia*, vol. XVI, Jul. 1992, pp. 193-214.

[12] Peter Wayner, "Strong Theoretical Stegnography," *Cryptologia*, vol. XIX, Jul. 1995, pp. 285-289.

[13] Mark T. Chapman, "Hiding the Hidden: A Software System for Concealing Ciphertext as Innocuous Text," The University of Wisconsin - Milwaukee, 1997.

[14] G. Hardy and E. Wright, *An introduction to the theory of numbers*, Oxford: Clarendon Press, 1960.

[15] *GLib*, Gnome: the free software project.

[16] S. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 35, 1987, pp. 400-401.

[17] P. Brown, R. Mercer, V. Della Pietra, and J. Lai, "Class-based n-gram models of natural language," *Computational Linguistics*, vol. 18, 1992, pp. 467-479.

[18] D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the I.R.E.*, September 1952, pp. 1098-1102.

[19] P. Meng, L. Huang, Z. Chen, W. Yang, and D. Li, "Linguistic Steganography Detection Based on Perplexity," *International Conference on Multimedia Information Technology*, 2008.

[20] C. M. Taskiran, U. Topkara, M. Topkara, and E.J. Delp, "Attacks on lexical natural language steganography systems," in *Proceedings of the SPIE International Conference on Security, Steganography, and Watermarking of Multimedia Contents*, January 2006.

VITA

Wesley J. Connell was born in Austin, Texas on June 3, 1977, the son of Kimberley Jeter and Tony Connell. He is the father of Azurean Teeple and husband to Kristy Peloquin. He received a Bachelor of Science in Applied Mathematics and Computer Science from Texas State University–San Marcos in May 2006, graduating Cum Laude. In August 2006, he entered the Graduate College of Texas State University–San Marcos.

Permanent Address: 5604 Southwest Pkwy #2114

Austin, Texas 78735

This thesis was typed by Wesley J. Connell.