The Pennsylvania State University

The Graduate School

College of Information Sciences and Technology

DETECTING MP3STEGO EMBEDDED CONTENTS: A STATISTICAL ANALYSIS ON A PARITY STEGANOGRAPHY ALGORITHM

A Thesis in

Information Sciences and Technology

by

Jonah Gregory

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2009

The thesis of Jonah Gregory was reviewed and approved* by the following:

Chao-Hsien Chu
Professor of Information Sciences and Technology
Thesis Advisor

Tracy Mullen
Assistant Professor of Information Sciences and Technology

William McGill
Assistant Professor of Information Sciences and Technology

John Yen
Professor of Information Sciences and Technology
Associate Dean for Research and Graduate Programs

*Signatures are on file in the Graduate School

ABSTRACT

Steganographic algorithms are designed to hide the existence of communications, and often do so by embedding hidden contents into innocuous covert media. Such forms of communication present a problem to security personnel and others concerned with detecting suspected groups' communication over the Internet. One tool available to conduct steganographic embedding in MP3 audio files is MP3Stego. In this thesis, we analyze the algorithm that MP3Stego uses, and study how MP3Stego capitalizes on the MP3 encoding algorithm to embed contents. We then dissect the MP3 file, and statistically analyze the lengths of the audio data blocks to test if we can differentiate MP3 files with MP3Stego embedded contents from those that do not. We compare our results to prior research, and find that the results do not agree with prior experimentation on MP3Stego. We then show and conclude that the best statistical predictor to examine when testing for MP3Stego embedded contents is the minimum value of the MP3 files' block lengths.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This thesis could not have been completed without the loving support of my daughter Elise, my parents Tom and Mary Gregory, my beloved grandparents, and my brothers, Micah, Jonathan, and Jess, but thanks go most of all to my wife Richelle, who was (and continues to be) so understanding of my long hours. Acknowledgements are due to committee members Dr. Tracy Mullen and Dr. William McGill, and to the diligent assistance and wisdom of my advisor, Dr. Chao-Hsien Chu.

I would also like to acknowledge Dr. Martin Ruckert for both his excellent book, *Understanding MP3*, and his advice and assistance with the technical details of the MP3 file format.

Chapter 1

Introduction

The rise and proliferation of the Internet, the "World Wide Web," has created a plethora of avenues for communication among colleagues, academia, business, and pleasure. While most of these communications use "normal" methods, such as email, web sites, and blogging, some forms of communication attempt to avoid discovery, and thus are designed to escape any attention. The science of communicating without alerting others to the existence of the communication is known as steganography.

Steganography is not a new science, and in fact documented instances of steganographic communication date back to Ancient Greece. However, the ubiquity of the Internet and the complexity of digital communications offer a powerful and accessible medium for hidden communication. In particular, modern compression algorithms have allowed for digital communication techniques that are extremely difficult to detect.

Most accessible steganographic programs utilize JPEG image compression to hide or embed content into carrier programs. Other encoding schemes offer other opportunities, and steganography is not restricted to image data. The MP3 encoding scheme has presented a medium for audio compression steganography, and the freely available program MP3Stego capitalizes on this opportunity.

Programs such as MP3Stego represent a conceptual proof of a powerful and complicated communications channel. Recent events have given rise to the belief, even fear, of certain malignant organizations using covert Internet communications to propagate and organize the efforts of terrorists, and even if these beliefs are unrealistic, understanding methods of steganographic communication present a worthwhile research problem. With this in mind, we

direct our research toward the mechanisms used to embed content with the program MP3Stego, and then conduct a statistical analysis to discover whether we can differentiate, via quantitative means, a meaningful method to separate files with hidden content from those that do not.

We are mindful of prior research involving MP3Stego. After assembling our experiment's control and hypothesis groups, we compare our statistical analysis with prior quantitative analysis, and find that previously found results that imply that the variance of the MP3 files' block lengths can detect hidden content do not agree with our results. We conduct statistical trials, and attempt to differentiate MP3Stego embedded content by the factors or variance, range, and finally the maximum and minimum values found for each sample. Our results indicate that the minimum value provides the best indication of MP3Stego embedded content, and screening our sample MP3 files by a minimum value threshold correctly segregates 99.5% of our sample set.

This study contributes to the awareness and understanding of steganographic techniques, primarily those that utilize advanced compression techniques. Because the MP3Stego program capitalizes on the advanced file format of the MP3 file and the complex encoding and compression techniques used to create MP3 files, gaining and understanding of how the program manipulates these processes will be useful for analyzing future steganographic algorithms, and attacking steganographic algorithms for which there is currently no known method of breaking the algorithm. Finally, our analysis provides a method with which to analyze MP3 files if MP3Stego embedded content is suspected.

This thesis consists of five chapters. Chapter one introduces the research problem and gives a very broad overview of the analysis vector, the software involved, and the contribution of the thesis research. Chapter 2 provides a brief analysis of the origin of steganography and information hiding, and describes prior steganographic programs, and also research regarding breaking and attacking such information hiding algorithms. We also discuss why the problem of

steganography is important, and why research regarding steganography is becoming more common. Chapter 3 begins by providing a more detailed view of the mechanisms involved in the encoding and creation of MP3 files. The basics of the MP3 file format are described, including the frame header, the side information, and the meaning of the various parameters of the file. The second section of Chapter 3 explains the concept of parity-based information hiding, the vehicle upon which the algorithm for MP3Stego is based. After parity-hiding is covered, the third section of Chapter 3 provides a detailed description of the process that is used by MP3Stego to hide an embedded file into an MP3 file during the encoding process. The final section of the chapter is a brief discussion of the strengths and weaknesses of the MP3Stego algorithm. Chapter 4 describes the procedures used in analyzing MP3Stego in this study. The first section describes the methods and location of the data that is analyzed. The next section describes the research hypotheses that we attempt to validate, and elaborates on why these hypotheses are meaningful to both the experiment we are conducting and the overall understanding of how MP3Stego changes an MP3 file when the hidden content is embedded. After the research hypotheses are set we discuss the metrics of evaluating the performance of our statistical predictors, and what indications the different types of success and failure provide. The final section of Chapter 4 provides the results of the data, our conclusions with respect to our hypotheses, and which predictor we feel is the strongest chance of detecting MP3Stego embedded content. The final chapter of this thesis summarizes the conclusions we have made.

Chapter 2

Related Works

In this chapter, the basics of information hiding are described. Beginning with a background of steganography, the art of covered writing, we present the concept and describe current methods of hiding information within data and digital media. We then outline why such techniques are pertinent to current security researchers, and why the problem of understanding MP3 steganography is important.

2.1 Background

The word steganography is of Greek origin. With a literal meaning of “covered writing,” the science of steganography is the concealing of the existence of a message. Where the aim of cryptography is to distort the meaning of a message to the point that it cannot be interpreted by an adversary or eavesdropper with any amount of effort; steganography focuses on ensuring that any adversary that happens to notice the communications medium will not know that a message or communication exists.

Steganographic communications are made up of several components: a cover text (or other carrier medium) within which any message is embedded, a stego-message that is hidden within the cover text, and the stego-key which is used to embed the message in such a way that it cannot be detected by an observer (Petitcolas, Anderson et al. 1999). Figure 2-1 shows a basic outline of the procedure of a steganographic algorithm (Westfeld and Pfitzmann 2000).

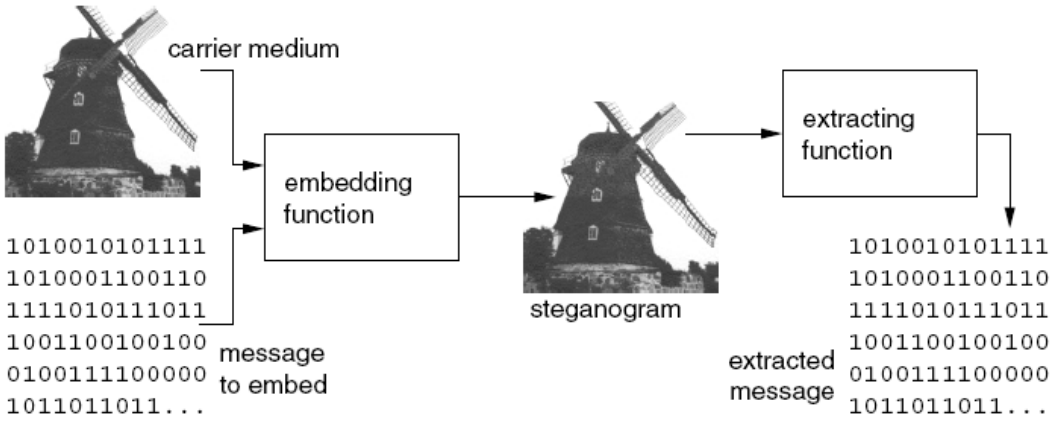


Figure 2-1: Outline of steganographic procedure (Westfeld and Pfitzmann 2000)

The roots of modern steganographic theory can be traced back to the year 440 B. C., according to *The Histories* (Herodotus 1992), when Histiæus tattooed the head of his most trusted slave with a message which disappeared when the slave's hair had grown back. Since then, many instances of hiding messages have been recorded.

2.2 Research in Steganography

One of the most influential works involving steganography is *The Prisoners' Problem and the Subliminal Channel* by Gustavus Simmons (Simmons 1984). In this work Simmons described some of the ideas and implications of steganography. These implications are described with the fictitious and well-known scenario "the prisoners' problem," which involves two prisoners (in separate jail cells) Alice and Bob, who are trying to send a message to each other. The prisoners have the ability to write messages to each other, but their warden "Willie" has the ability to read any such message before delivering it to the other prisoner. Simmons discusses the entropy involved in sending hidden messages, and the increasing chance of message discovery as the amount of cover medium increases. Hiding a message also requires that some degree of

predetermined information is known by all parties involved in sending and receiving the message, much like cryptography. Another important concept discussed in this work is the idea of an "active" or "passive" warden. Essentially, this means that the techniques and strategies used by parties change dramatically depending on the level of attention paid by the warden examining the messages. If this warden does not suspect that the two parties are exchanging any information, or rather are exchanging any information besides what is obviously written, the act of exchanging hidden messages will be much simpler than if the warden suspects secret communication and is actively searching for something of that nature. Further, this may prompt a warden to change parts of a cover medium in the hopes of uncovering or destroying a hidden message. Of course, once the warden is aware of any communication, it is within his power to prevent the communication, or to put forward that he is unaware of the communication and then attempt to decipher and possibly purposefully change it.

Perhaps ironically, Simmons' work had a hidden message as well as the motivation and implementation of steganography. At the time, the United States and the USSR were discussing the placement of sensors in each others' nuclear facilities in the interests of nuclear arms treaties. Simmons' paper had the true intent of drawing attention to this event (Anderson 1996). Simmons' work was expounded in the next decade with several works. *Stretching the Limits of Steganography* by Ross Anderson introduces the concept of public key steganography, and also compares the inverse analogy between cryptography and steganography further; one of the questions introduced is whether or not there exists a proof of perfect covertness in the sense that a one-time pad can be proven to provide perfect cryptographic secrecy (Anderson 1996). Two years later, Anderson and Fabien Petitcolas' *On the Limits of Steganography* provided a more comprehensive and theoretical value of steganography and its applications. It is in this paper that Petitcolas first introduces the concept of the "power of parity," which is the concept which

directly relates to this thesis, although parity was mentioned in Anderson's prior work (Anderson 1996; Anderson and Petitcolas 1998).

Since the rise of networked computers and the Internet, methods of hiding information and communication have been devised, implemented in shared. Simple methods, such as replacing the least significant bits of simple graphic files, can be considered simple compared to newer methods which can hide bits in the construction of advanced techniques such as JPEG graphic compression or audio and video codes. Some schemes even use simple phrases which change depending on which bits need to be hidden, yet make sense no matter which phrase is used.

The program Jsteg is a readily-available program that can hide stego-content within a JPEG image file (Upham 2009). One of the first JPEG hiding programs, it was followed by many others, including F5, JPHide/JPSeek/JPHSWin, EzStego, Steganos, S-Tools, and the more recent StegHide, are some of the more popular and accessible steganographic programs, but there are many more available via the Internet. Although the algorithms are often able to able to be compromised and the hidden content extracted in the original forms (Westfeld and Pfitzmann 2000), in many cases the source code is readily available and modifications and improvements are possible.

However, no matter the complexity of a steganographic scheme, the true intent is the hiding of the message, and if that goal is accomplished, the communication is more secure than if it were encrypted with the most sophisticated algorithms known to man.

2.3 Research in Steganalysis

Attempts to detect steganographic communications on the Internet have largely been unsuccessful. A study conducted by Neils Provos and Peter Honeyman concluded that a search of

two million images from eBay revealed not a single steganographic embedded message using algorithms designed to detect the most common and readily available forms of image steganography (Provos and Honeyman 2001). Part of the reason that such analysis is difficult is the realization that the suspicion raised from a false positive in detection can provide more misleading results than a false negative.

Although efforts to detect hidden digital communication have not been fruitful, this does not imply that such a carrier medium is not accessible or powerful. It becomes all even more important to understand current steganographic methods and media. In this study, we examine the use of parity-based steganographic communications through the use of a common audio file, MP3. Fabien Petitcolas' MP3Stego program is freely available through the Internet, and provides a powerful method of embedding hidden message content into the audio blocks of an MP3 file without damaging the audio quality so much that the average listener will detect the change (Petitcolas 1997-2009).

A notable steganalysis work by Andreas Westfeld, "Steganalysis in the Presence of Weak Cryptography and Encoding" (2006), reveals that some steganographic algorithms can be broken with respect to excluding innocuous cover files. Out of 18 steganographic algorithms analyzed, only two algorithms can be completely detected, and 15 out of 16 can be broken with respect to excluding innocuous cover files. The only steganographic algorithm in this study which is not listed as being able to be broken is MP3Stego, the focus of this research.

MP3Stego is not the first steganographic algorithm designed for audio files. Methods have already been devised and implemented, including LSB methods, hiding with phase and amplitude modification, and other MP3 transformations in the VLC domain (Cvejic and Seppanen 2002; Gopalan, Wenndt et al. 2003; Tan, Wang et al. 2008). Although these other methods of MP3 steganography have been devised, conceptual proof-of-concept programs are not as accessible as MP3Stego, one reason that MP3Stego's algorithm must be addressed. The

popularity and ease of access of MP3Stego makes this program an important steganographic methodology to understand and perhaps even defend against.

In recent years the threat of terrorism and the rise of a worldwide communications network has elevated the suspicion and increased the importance of detecting steganographic communications. Petitcolas et al shows the increasing amounts of attention given steganography in *Information Hiding – A Survey* by showing how the amount of publication has risen exponentially throughout the 1990's (Petitcolas, Anderson et al. 1999). Especially since the terrorist attacks in New York on September 11, 2001, there has been a great deal of suspicion regarding terrorists and the use of steganographic media to orchestrate terrorist attacks, of which only a few articles are cited here (Kelley 2001; McCullagh 2001; Schneier 2005). Because of this, and the ease of communications facilitated by the World Wide Web, understanding methods of steganography and information hiding is an important research problem, and the basis for this experiment.

2.4 Other MP3 Steganographic Algorithms

Other methods exist for embedding hidden content in MP3 files. In this section, we discuss two algorithms that are available via the Internet. These algorithms are named Mp3stegz and UnderMP3Cover.

The first, Mp3stegz, utilizes slack space within the Mp3 file to embed data (Zaenuri, 2009). After a hidden file is chosen, the file to be hidden is encrypted and separated into pieces. Each frame in the MP3 file has a limited amount of slack space. Unlike Mp3Stego, which uses the actual encoding process to aid in the embedding process, Mp3stegz takes an existing MP3 file and overwrites data that it deems unimportant to the quality of the MP3 file. Although the bulk of the file data is encrypted, portions of the control data remain in plaintext. This is done so that the

length of the hidden content can be accurately judged, and the subsequent decryption of the hidden file can be executed without error.

Although Mp3stegz' algorithm can be effective in transmitting hidden context, it is simple to detect the hidden content. Simply viewing the MP3 file in a hex editor will reveal the hidden content: in addition to the content overwritten into the cover MP3 file in plaintext, the existence of the data does not coincide with the standard MP3 file format. It is important to understand that detection of the hidden content that Mp3stegz embeds in to the MP3 file does not represent a complete break of the hidden information. Even though the existence of the hidden message can be shown by examination, the attacker still has to defeat the cryptography which protects the embedded file. In order to do so, the attacker must extract the exact data of the encrypted file, deduce the algorithm with which the file was encrypted (although knowledge of the Mp3stegz algorithm will reveal this), and use computational brute force to extract the password (and possibly the initialization vector) with which the hidden content was scrambled. This will not prove an easy task. However, with steganographic communication, discovering the existence of the hidden message is still a serious breach, because of the implications of steganography in general. Unlike encryption, which hides the message but not the communication, steganographic communication attempts to hide the existence of the message itself, which implies that the sender believes that there will be serious consequences if the communication is discovered. Once the hidden communication is detected, the detector will certainly wonder what sort of message requires that no party is aware that two entities are communicating. Steganographic discovery can then lead into further research into who is attempting to secretly communicate with whom, and for what reasons. Such implications can be more important than the actual contents of any one specific message.

Like Mp3stegz, UnderMP3Cover utilizes existing MP3 files to embed steganographic content (Platt, 2009). UnderMP3Cover is also available via the Internet. While UnderMP3Cover

uses an algorithm that has some similarity to Mp3stegz with respect to overwriting an existing cover MP3 file, the algorithm that it uses is more complex. Instead of overwriting large sequential blocks of information in the file, UnderMP3Cover writes bits sporadically throughout the MP3 file's blocks, using small control blocks written in the side information section of the MP3 frame. Overwriting data that will not greatly impact the quality of the cover file is called Least Significant Bit (LSB) steganography. These algorithms can be successful if there is no reason to suspect steganographic content, but research has shown that even a more complex algorithm of this nature can be vulnerable if an attacker decides to analyze a cover MP3 file, with UnderMP3Cover and other LSB steganographic methods (Westfeld, 2006).

Chapter 3

MP3-Based Steganography

In this chapter, the basic mechanism of the MP3 encoding process is described, so that the algorithm that MP3Stego uses to embed hidden content can be better understood. After a short description of the MP3 file, the basic concept of parity-based information hiding is described. After understanding parity-hiding, the mechanisms used in MP3Stego are described in detail.

3.1 A Primer in MPEG Audio

MPEG-1 Audio Layer 3, or MP3 audio, is a pervasive digital format. Designed by the Moving Picture Experts Group (MPEG), MP3 uses a lossy form of audio compression to greatly reduce the size required to store audio information. MP3 files are created by using an encoding application to transform an audio track into an MP3. In addition to using advanced compression techniques, acoustic techniques such as tone masking and signal compression based on human perception models are used to achieve maximum size-to-quality ratio. (Ehmer 1959; Jayant, Johnston et al. 1993) The quality of the MP3 is further configurable by the amount of information allocated to the file when it is encoded; this is known as the bit rate of the MP3 file. A standard bitrate is 128 kilobits per second, but bitrates ranging from 32-320 kilobits per second are standardized.

Using MP3 audio encoding, the size of an audio file can be reduced to about $1/10^{th}$ of the size of the same file with no compression. Although MP3 audio may not be considered “high-fidelity” by audio experts, due to the strategic removal of some frequency bands, the size and convenience of use of the MP3 audio made the format extremely popular, beginning in the mid-

1990's. This ease of use, coupled with the rise of the Internet, the MP3 player WinAmp, and the insufficient structure of protection of music copyright, made MP3 usage spread at an exponential pace through the 1990's, and with the revolutionary launch of the first major peer-to-peer sharing network, Napster, in 1999, the MP3 became the de facto standard for portable music (Wikipedia 2009). Although this diverges from this subject matter, it is important to note that this phenomenon directly contributed to the pervasiveness of the MP3 file, and this quality of MP3 is part of what makes this particular file format, along with the larger average file size, an excellent candidate for embedding and transmitting hidden content.

MP3 files use a different format than most files with regard to headers, trailers, and file data. Usually, a file will begin with a header or control block section, followed by the data of the file, with some file types concluded by a trailer containing additional file data. Breaking from this format, MP3 files are organized into sections designated "frames," which can be abstractly thought of as a smaller version of the standard header/data format. As the audio data is encoded into an MP3 file, frames are created and the audio data is encapsulated into the frames. Each frame begins with a 32-bit header, which is used to synchronize the audio playback and specify characteristics of audio encapsulation; including bitrate, sampling rate information, and a flag bit that signals the presence of lack of a cyclic redundancy check within each MP3 frame for quality control.

The frame header of an MP3 file contains important technical information regarding the frame. Important information that can be found in the frame header includes:

- The frame sync block that enables the program that decodes the MP3 file to synchronize the bit stream with the MP3 file
- The MPEG Audio version ID and Layer descriptor that is used to encode the MP3 file

- Indicator bits to specify the presence of a cyclic redundancy check, padding bits, and copyright.
- Audio information, including bitrate index, and stereo/mono channel mode.

The information written in the frame header is dynamic depending on the encoding algorithm used to create the MP3 file. Some encoding programs will create MP3 files with frame information will not change at all, while others may have frames that change often, usually in an attempt to compress the audio data further or increase the quality of the audio. The ability of the MP3 file format to change the audio compression and sampling rates dynamically through the playback of the audio data is one of the strengths of the MP3 format. Figure 3-1 shows an abstracted view of the MP3 frame header with the names of the data fields within the header.

Once the frame header is found and interpreted by an MP3 software application, the header is discarded and the data encapsulated in the frame is examined. This data is prefaced by another block of control information, called the “side information.” From the side information, characteristics including the amount of data in the block and the channel mode (stereo/mono) of the block are denoted before extracting and decompressing the actual audio data. An abstraction of the format used in MP3 audio files can be seen in Figure 3-2. Interestingly, an MP3 file can be split on any frame boundary, and both pieces will play as fully functional audio files, since all of the necessary information is contained locally within the frame header and side information.

The details of the actual audio compression and decompression are very complicated, and are out of the scope of this research. It is enough for the reader to understand that quantization is used to transform the range of audio signal values to a set of discrete values. After this is complete, the perceptual analysis and compression techniques are applied, and the resulting audio data is stored in the frame after the side information. Each frame contains two “blocks” of audio data per channel, for a total of two blocks for a mono-channel MP3, and four for an MP3 in stereo

(2 channels). A detailed description of the techniques, mathematics, and algorithms used in this process can be found in Martin Ruckert's book *Understanding MP3* (Ruckert 2005).

In addition to the control and audio portions of an MP3 file, a small block of ASCII-encoded information is usually attached to the beginning or end of the MP3 file. This "tag" contains information about the audio content of the MP3 file, and is divided into fields. The standard for MP3 tags is now the ID3v2 tag, although ID3v1 is still commonly found in use (2009). Common fields found in the ID3 tag include the artist who recorded the song, the album the song is found on, the year in which it was released, and the genre of the music.

Bits	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Binary	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	0	0	0
Hex	F				F				F					B			A					
Meaning	MP3 Sync Word												Version	Layer		Error Protection	Bit Rate				Frequency	
Value	Sync Word												1 = MPEG	01 = Layer 3		1 = No	1010 = 160				00 = 44100 Hz	

23	24	25	26	27	28	29	30	31	32
0	0	0	1	0	0	0	0	0	0
0		4						0	
Pad. Bit	Priv. Bit	Mode		Mode Extension (Used With Joint Stereo)		Copy	Original	Emphasis	
0 = Frame is not padded	Unknown	01 = Joint Stereo		0 = Intensity Stereo Off	0 = MS Stereo Off	0 = Not Copy-righted	0 = Copy Of Original Media	00 = None	

Figure 3-1: Abstracted layout of the MP3 frame header. (Wikipedia, 2009)

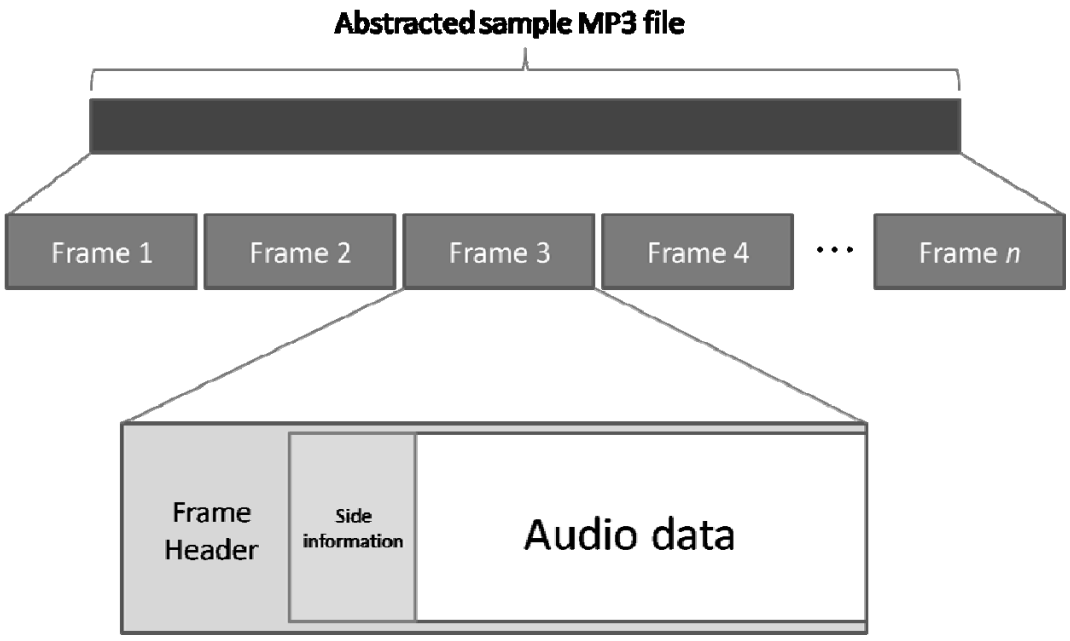


Figure 3-2: Abstracted diagram of the MP3 file format.

3.2 Parity-based Hiding

Where the more common steganographic technique of least-significant bit (LSB) hiding seeks to directly replace bits that will likely escape notice with different bits that make up a 'hidden message,' parity-based hiding instead seeks to use the properties of a block of data to embed a hidden message. Specifically, the parity of a certain delimited block of data is used. The parity of a block of data is the sum of the bits in the block modulus 2, or more simply, whether the sum of the block is even or odd. The resultant bit for that block of the hidden message is determined by this parity. The "power of parity" was first described by Anderson and Petitcolas (Anderson and Petitcolas 1998).

Let's use an example to demonstrate how such a scheme would be used to hide a message. Assume for this example that the blocks of data are somehow controlled by the creator and can be arbitrarily recompiled without changing the cover file's contents, are 8 bits long, and

that we are hiding the message “h” into a file. We begin with the sample file shown in Figure 3-3 (a).

First, we examine the message that we want to hide. In this case, it is the ASCII letter “h”, which is represented by the number 104, written “01101000” in binary. Now that we know the bits we need to create the hidden message, we examine the current parity of the blocks in the file. The parities of the byte “blocks” are shown in Figure 3-3 (b).

By extracting the parity of the blocks and assembling them independent of the file contents, we create the message “00100011”, which is not quite the message we want to send. But if we use our method of arbitrarily recompiling blocks of our cover file on blocks 2, 5, 7, and 8, until we get our desired parity, we can change our cover file until it looks like the sample in Figure 3-3 (c).

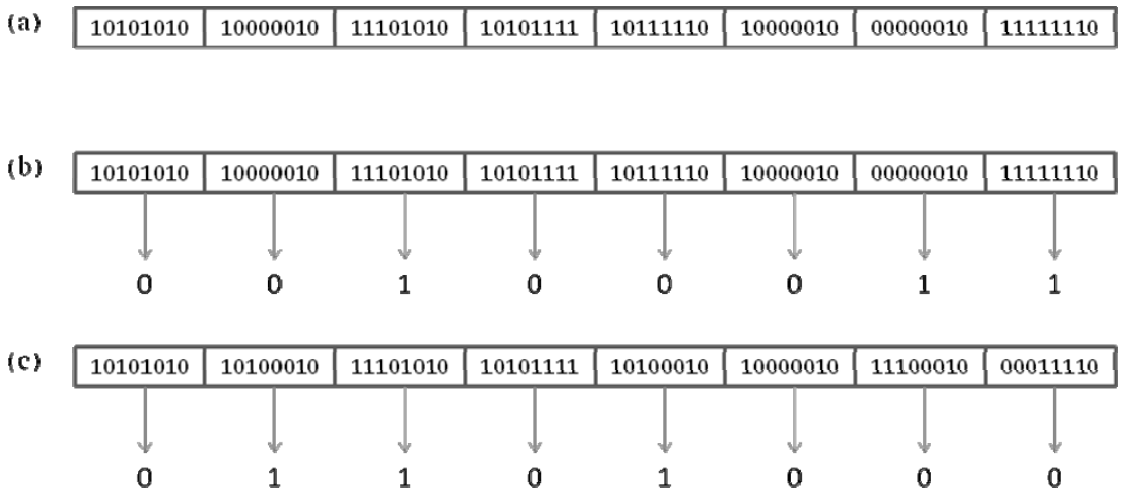


Figure 3-3: (a) A sample binary file, divided into bytes; (b) the calculated parities of each of the bytes of the sample file; (c) rearranging bits of the sample file allows the parities of the samples’ blocks to be manipulated.

Now the blocks of the file, when examined for parity, reveal the hidden message's bits without outwardly showing the existence of the message. While it is easy to see the merits of this type of hiding scheme, it is also trivial to see a few stipulations and restrictions regarding parity-based hiding schemes:

(1) First, and most glaring, is this miraculous ability to "recompile" blocks at will. We can see that such an action makes many normal file types, especially simple file types such as ASCII text, immediately unsuitable for this type of hiding strategy. Such a recompile would most certainly destroy the integrity of the cover file, if not transforming it into outright garbage. This restriction makes only certain types of files, most notably those that use data compression or encoding, suitable for this type of transformation.

(2) Such a strategy must have a defined block size. Although different embedding tools could simply make this an arbitrary size, this would lead to much simpler detection methods once the existence of the scheme is revealed to an attacker.

(3) It is also apparent that we are using a rather large cover-to-hidden data ratio. While our example uses only 8 bits for each hidden bit, in application such a ratio would likely make statistically significant tracks, making detection quite simple. Because of the large ratio, the cover file will also have to be a rather large file on average to support hidden messages of substantial size.

(4) Encoding to hidden messages straight into ASCII text would make the parity hiding scheme easy to detect. This is rather obvious, and rather trivial; simply encrypt the message before hiding it. This also requires more computation, and the use of a key to recover the hidden message. If the sender could be assured that no attacker would be scanning for parity in the cover file, it may not be necessary, but such assurances normally cannot be made.

(5) How do you know when to stop decoding parities? Of course, if the hidden message were encoded into standard ASCII text, it would make sense to stop decoding parity when the message stops making sense (e.g., stops translating into viable ASCII). However, as we mentioned in (4), this may not be a viable method. If we encrypt our hidden content, it becomes crucial that the decoder knows when to stop adding bits to the message to be decrypted – adding extra bits will make the hidden message undecipherable even with the key because the decryption algorithm will fail. This restriction makes it necessary to embed a control block separate from the actual hidden content, to let the recipient of the message know the message length. It is feasible, however, to either encrypt this control block with the same key as the message, to preserve confidentiality, or even embed it with a different hiding scheme.

We can see that parity-based data hiding can be effective, but has a few strong stipulations that make it very difficult to use in practice. Essentially, it needs the perfect type of cover file to hide the weaknesses of the strategy. In the next section, we examine the tool MP3Stego, a parity-based hiding tool designed to use the MP3 file format to accomplish this goal.

3.3 Mechanism of "MP3Stego"

MP3Stego, devised and implemented by Fabien Petitcolas (Petitcolas 1997-2009), is an addition to the source code of the 8hz MP3 encoding engine (8hz Productions 1998). MP3Stego utilizes the compression techniques, large size, and pervasive nature of MP3 files to create a steganographic medium using a parity-based embedding scheme.

On the program website, the author describes the operation of the MP3Stego program thus:

"MP3Stego will hide information in MP3 files during the compression process. The data is first compressed, encrypted and then hidden in the MP3 bit stream. Although MP3Stego has been written with steganographic applications in mind it might be used as a copyright marking system for MP3 files (weak but still much better than the MPEG copyright flag defined by the standard). Any opponent can uncompress the bit stream and recompress it; this will delete the hidden information – actually this is the only attack we know yet – but at the expense of severe quality loss.

The hiding process takes place at the heart of the Layer III encoding process namely in the inner_loop. The inner loop quantizes the input data and increases the quantiser step size until the quantized data can be coded with the available number of bits. Another loop checks that the distortions introduced by the quantization do not exceed the threshold defined by the psycho acoustic model. The part2_3_length variable contains the number of main_data bits used for scale factors and Huffman code data in the MP3 bit stream. We encode the bits as its parity by changing the end loop condition of the inner loop. Only randomly chosen part2_3_length values are modified; the selection is done using a pseudo random bit generator based on SHA-1.

We have discussed earlier the power of parity for information hiding. MP3Stego is a practical example of it. There is still space for improvement but I thought that some people might be interested to have a look at it." (Petitcolas 1997-2009)

Without a thorough understanding of the algorithms used to encode MP3 files, this description is difficult to understand. One concept that is crucial to the MP3 encoding process and the MP3Stego algorithm is quantization. Quantization is the process of converting a continuous analog waveform into a set of discrete digital values. Discrete values are created at a set time interval, called the sampling rate. Sound waves are naturally analog waveforms, and so to convert those to digital media for computers and digital devices to use and store, it is necessary to perform quantization on the sound file's analog waves. Quantization is also the step of the MP3

encoding process that is done and redone as needed to acquire the block parities needed to hide information within the MP3 file. Figure 3-4 shows an abstraction of the quantization process.

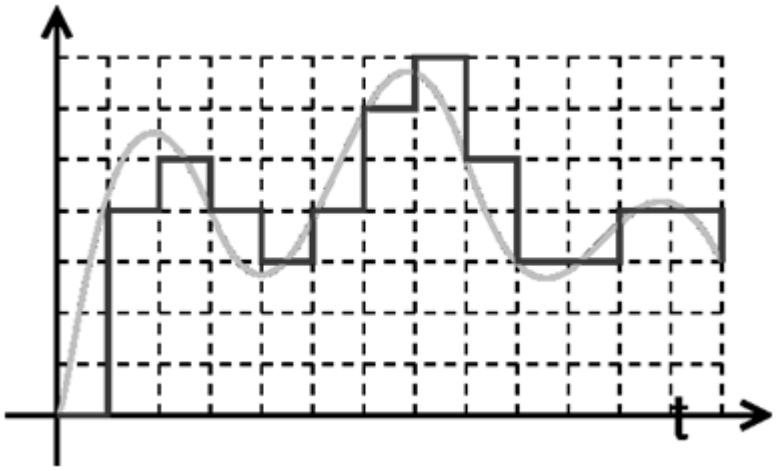


Figure 3-4: An abstraction of the process of quantization. The analog waveform is converted into a set of discrete digital values. The *x*-axis in this figure represents time, and the marked increments represent the sampling rate. One discrete value is created at each sampling interval. (Wikipedia, 2009).

Now that we've introduced the concept of quantization, we can elaborate more on the MP3Stego algorithm. To clarify the MP3Stego algorithm, the process can be generalized to these steps:

(1) A lossless audio file is chosen to be converted to an MP3 file with the MP3Stego-enhanced 8hz MP3 encoding application. This application is currently downloaded as Encode.exe and its opposite, Decode.exe. Currently, the only format of audio file that this application will work with is the MS-RIFF WAVE audio format (.wav).

(2) The application runs under a DOS command-line. Along with the executable name, the user may pass a switch that tells MP3Stego to attempt to embed a file into the new MP3. If the user passes this flag, they must also pass the password used to encrypt the embedded file, and the filename of the MP3 output file.

(3) Once the executable is run with the correct command-line parameters, MP3Stego examines the source audio WAVE file to see if it will have enough space to embed the entire hidden message. The program fails if the source WAVE file is too small, and MP3Stego tells the user the bit limit for that particular file.

(4) If all parameters are in order, and the source file has sufficient space to embed the hidden file, the hidden file is compressed and encrypted, and the bit stream of the hidden file is prepared for embedding into the MP3 file. MP3Stego begins encoding the source audio file into an MP3 file, but inserts several steps into the normal encoding process:

(5) For each block of music information to be encoded, MP3Stego decides whether or not or encode a hidden bit of information. Two factors decide whether to hide a bit: if there are more bits in the hidden message yet to be encoded and whether this particular block has been randomly chosen to be embedded with a bit.

(6) If it has been determined that no bit is to be embedded, no change is made from the standard encoding process, and the block is created as normal.

(7) If a bit is to be embedded, the block is first encoded as normal. After the initial encoding, the parity of the block is examined. Once the parity of the block is determined, one of two scenarios will occur:

    a. The parity bit matches the bit to be encoded. In this case, since the parity bit is what we want it to be, the initial encoding is suitable, since it produces the correct bit for the hidden message. No change is made and the initial encoded block is placed in theMP3 file.

    b. The parity bit doesn’t match the bit to be hidden. In this case the block is unsuitable for carrying the hidden message bit because the parity is incorrect. The quantization factor that is used by the encoding application is changed, and the block is re-encoded in an attempt to change the parity bit of the block. This

may happen several times before the parity of the block is correct. This also changes the length of the block.

(8) If the file cannot embed the file for some reason, MP3Stego fails. If there are no errors, once the hidden file is completely embedded, the remainder of the MP3 file is encoded normally.

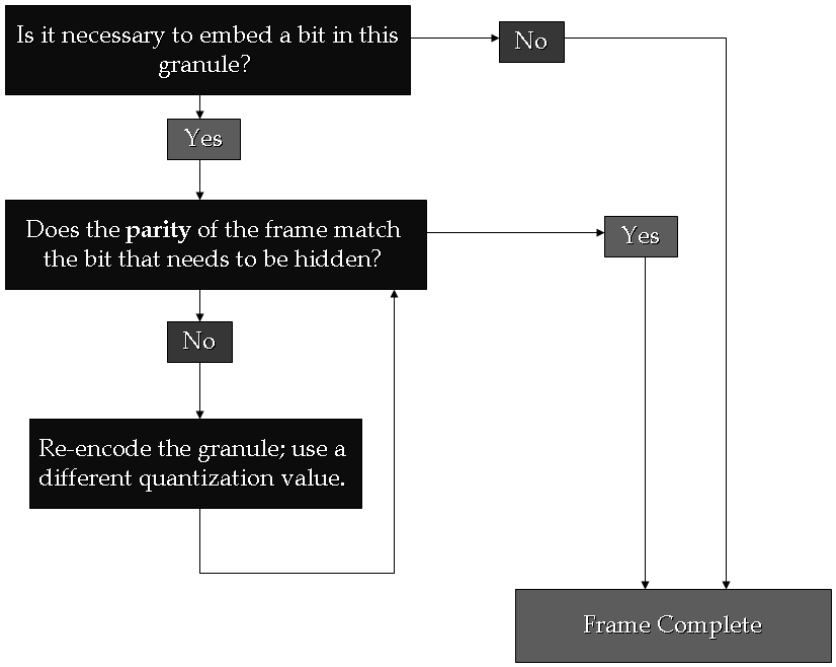


Figure 3-5: Abstracted flowchart of the frame encoding segment of the MP3Stego embedding algorithm. For each encoding step, MP3Stego decides whether a bit needs to be hidden. If one does, parities are checked and the encoding process is redone with a different quantization value if the parity bit does not match the bit to be hidden.

The section of the algorithm that deals with deciding whether to hide a bit and checks the parities is abstracted and shown in Figure 3-5.

MP3Stego also includes a decoding tool, Decode.exe, which will decode an MP3 file into a pulse-code modulated audio file (.pcm), and attempt to extract an MP3Stego-embedded file if a flag and password are provided. This operation is essentially the inverse of the encoding operation.

3.4 Strengths and Weaknesses in MP3Stego

We can see that the design of MP3Stego, combined with the MP3 file format, overcomes the major obstacles of using a parity-based hiding scheme. Encoding an MP3 file enables the data blocks to be changed in such a fashion that it becomes possible to execute the process several times when the parity of the block is unacceptable. The MP3 file format already has a defined block size which is suitable for using for parity-checking. The pre-existence of a block format also means that an attacker may be less likely to notice the blocks with respect to parity, since they are supposed to be there. The large amount of data in the average MP3 file provides an excellent habitat for the large cover-to-hidden data ratio used in parity-based hiding schemes. MP3Stego uses compression and encryption to hide any traces of obvious embedding. Lastly, MP3Stego embeds a control block to let the decoder application know where and how to retrieve the hidden file. All these factors contribute to providing a suitable medium for using such an embedding scheme.

MP3Stego allows for a viable steganographic attack using common files. In the next chapter, we will attack the methods used by MP3Stego and determine if a statistical attack on an MP3 file can reveal the presence of a hidden embedded message.

In “Detecting low embedding rates,” Dr. Andreas Westfeld examines the MP3Stego algorithm for methods of detecting embedded content in MP3 files (Westfeld 2003). Westfeld

arrives at some fruitful conclusions; namely, that the changing of the quantization factors when encoding the MP3 file with the MP3Stego algorithm, and the consequent changing of the block lengths, will produce statistically significant changes in variance in the block lengths of the MP3 file. In this section, we will take a closer look at the methodology used to detect embedded content in this fashion.

Additionally, Westfeld demonstrates the use of an algorithm to detect the encoding application used to create the MP3 file will provide a domain or subset of files which cannot contain MP3Stego embedded content. The reasoning here is that since the MP3Stego application uses the 8hz MP3 encoder, if an MP3 is determined to been created using another encoder, it cannot then be embedded with MP3Stego content, since MP3Stego was not used to create the file. Further exploration down this path can be found in the paper "Statistical Characterization of MP3 Encoders for Steganalysis: CHAMP3" (Böhme and Westfeld 2004).

This research may prove fruitful for detecting which encoder was used to create an MP3 file, and therefore be useful in ruling out a subset of files which cannot be embedded with MP3Stego hidden content. However, we want to examine the actual properties of the file more closely, with the objective of pointing out statistically significant differences within the file. With success, we would be able to determine if MP3 files were embedded with hidden content should the algorithm from MP3Stego (or a comparable algorithm) were incorporated into a different MP3 encoding engine.

Looking at the results displayed in "Detecting low embedding rates," Westfeld shows in Figure 3-6 that the variance of the resultant MP3 files can be predicted from a function involving the size of the file and the amount of payload. This would indicate that we should be able to obtain an estimated variance value by using these two predictors. However, the amount of payload would not be a good predictor in a real-world detection scenario because it would obviously be unknown. Without that value, such a prediction would be considerably more

difficult. There is a line drawn where the variances do not overlap; this means that if we take a sample of MP3 files of comparable size, we should be able to use variance as a predictor to determine whether there is MP3Stego embedded content within a file.

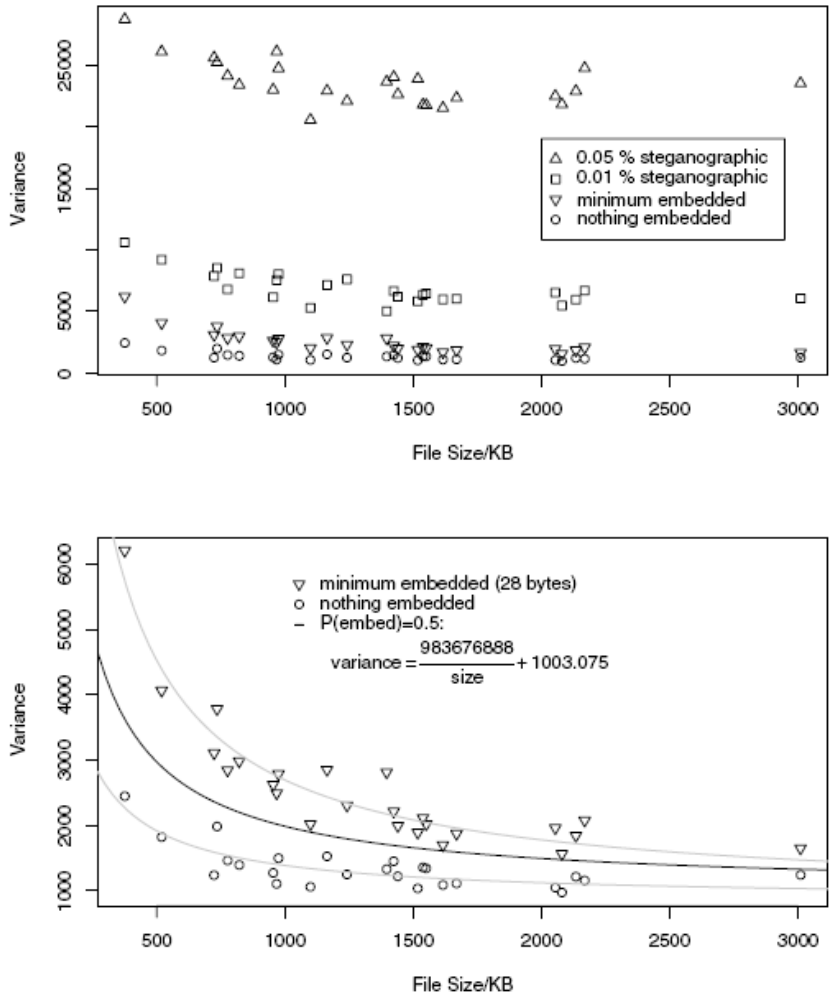


Figure 3-6: Variance to File Size of MP3 files as shown from “Detecting low embedding rates” (Westfeld 2003)

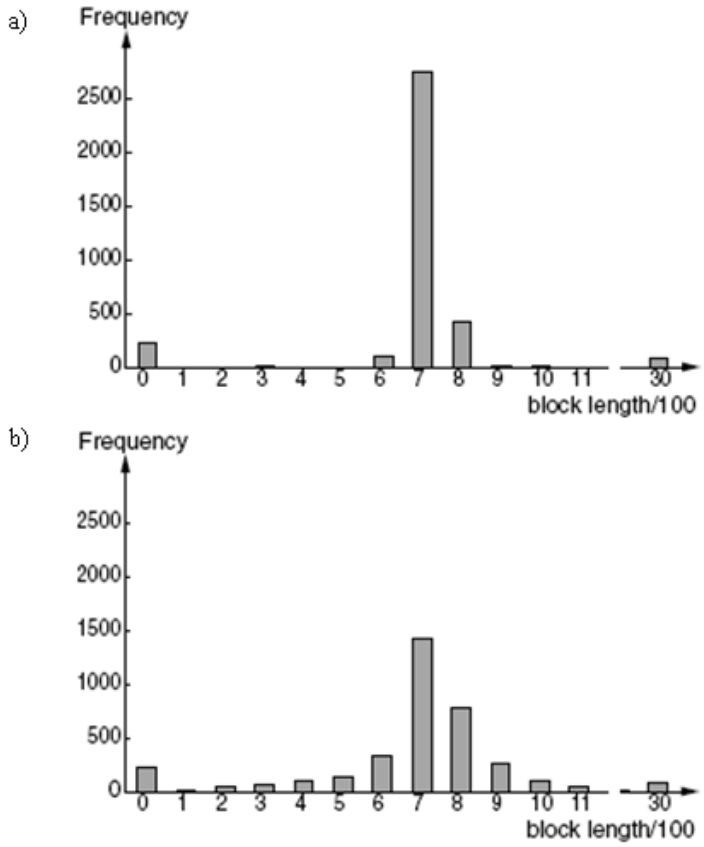


Figure 3-7: Histograms of the MP3 block lengths from "Detecting low embedding rates" (Westfeld 2003). The histogram (a) is from an MP3 file with no embedded content; the histogram (b) is created from an MP3 file with an embedded file.

In Figure 3-7, two histograms are shown, comparing block length distribution of a normal file and a file with embedded content. By examining the histograms, it follows that the variance of the two files' block lengths would be different; but the information provided from the two histograms does not allow for a serious analysis. Are the two histograms from the exact audio source? Do they represent all such MP3 files?

A further statistical analysis will provide answers to these questions. In the next chapter, we convey the results of our statistical analysis.

Chapter 4

Experimental Design and Analysis

In this chapter, we outline the design of the experiment we use to determine the efficacy of statistical attack on MP3Stego. First, we begin by describing the construction of control and experimental sample groups. We then outline our research hypotheses. After that, we describe how to measure the performance of each of the groups. Finally, we discuss which measures proved to be the most effective given the results of the trials.

4.1 Data Source and Experimental Design

In order to analyze the effects of MP3Stego on encoded MP3 files, we first assemble a set of audio source files to experiment with. We randomly selected 50 WAVE audio files. These audio tracks are from all different genres of music, and vary in musical intensity, loudness, tempo, and in most other respects.

Limiting length should provide some control over the variance analysis. Because message length will impact MP3 files of different length to different degrees, we conduct our analysis with a static length. Once we can conclude some results, application according to length could be a further analysis. To implement this, we restrict all of our audio source files to the 21.5 – 22.5 megabyte range. This range of size for WAVE source file will translate to the range of 3.9 – 4.2 MB when encoded to MP3. This range is narrow enough to restrict length variation, yet still representative of an average MP3 file size. Additionally, each WAVE audio track is one channel only, or “mono,” for purposes of having a comparable amount of blocks in each file.

As we mentioned, the length of embedded message with respect to the cover MP3 file will have impact on the total variance, and possibly other relevant statistical data. To account for this in our analysis, we need to perform the test on MP3 files with varying amounts of hidden information. To track the effect of increasing the message size, we embed three different message sizes into each audio source file, where each different message size is named for its relation to the source file's hidden content capacity in terms of size – denoted as {small, medium, large} and the encoded MP3 with no embedded content is denoted as {none}. So, for each audio source file, named by number: {1.wav, 2.wav, 3.wav … *n*.wav}

We create, by encoding with the MP3Stego program, a 'set' of four MP3 files:

{{1-none.mp3, 1-small.mp3, 1-medium.mp3, 1-large.mp3},

{2-none.mp3, 2-small.mp3, 2-medium.mp3, 2-large.mp3},

{3-none.mp3, 3-small.mp3, 3-medium.mp3, 3-large.mp3},

…

{ *n*-none.mp3, *n*-small.mp3, *n*-medium.mp3, *n*-large.mp3}}

Table 4-1: Description of the creation of MP3 sample sets.

Source Audio File	MP3 with No Embedded Content	Small Embedded Content	Medium Embedded Content	Large Embedded Content
1.wav	1-none.mp3	1-small.mp3	1-medium.mp3	1-large.mp3
2.wav	2-none.mp3	2-small.mp3	2-medium.mp3	2-large.mp3
3.wav	3-none.mp3	3-small.mp3	3-medium.mp3	3-large.mp3
…	…	…	…	….
50.wav	50-none.mp3	50-small.mp3	50-medium.mp3	50-large.mp3

We now have a total of 200 sample MP3 files organized into 50 sample sets. The creation of the sample file set is described in Table 4-1. Table 4-2 shows the actual hidden messages embedded within the four different groups of files, along with the sizes of the messages. Figure 4-1 provides an abstraction of the sample files created for this experiment.

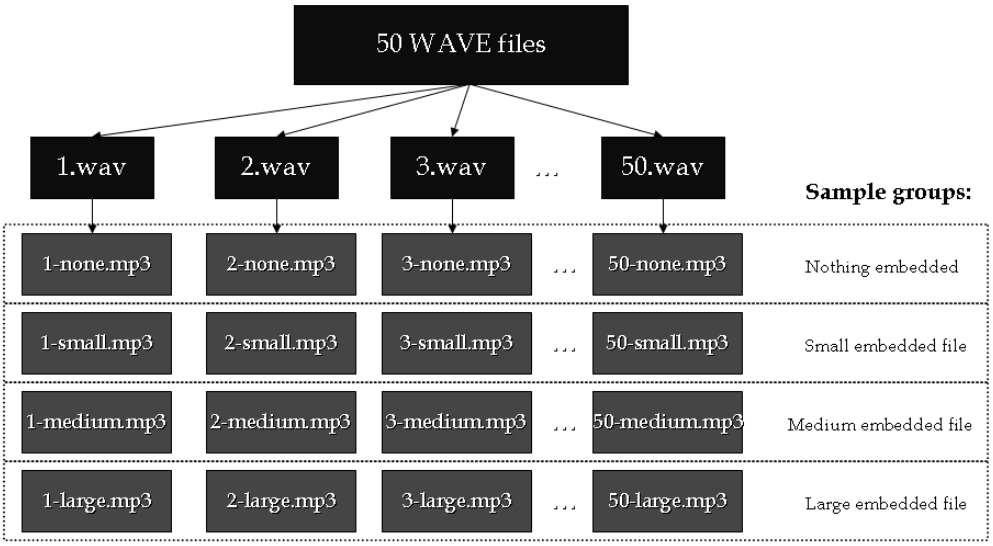


Figure 4-1: Abstraction of the sample file set.

Table 4-2: Messages embedded into the four sample groups.

Sample Group	Embedded Message	Size of message
none	none	-
small	“Hello world!” (17 times)	206 bytes
medium	“Hello world!” (105 times)	1.23 Kbytes
large	“Hello world!” (203 times)	2.37 Kbytes

Now we can track statistical changes relative to a non-embedded, “clean” encoded MP3 file. First, we must extract the relevant data from the files. We are dealing with block length, meaning the value of the blocks of audio data in each MP3 frame. For each channel in the MP3

file, there are two blocks. Since all test files are created as mono, single-channel MP3s, we will extract two block lengths per frame. The value of the block length is stored in the *main_data_bit* field in the side information section of the MP3 frame, and there is a *main_data_bit* field for each of the blocks, so we gather the two block lengths per frame by extracting these two fields from the side information (Ruckert 2005). We extract the *main_data_bit* data by creating a small program from the C++ language, compiled on the g++ compiler, that locates the frame sync block of each MP3 frame, then calculates the correct location of the *main_data_bit* fields and extracts and outputs the values to a tab-delimited text file, which is converted to spreadsheet format. The spreadsheet data is then input into the Minitab statistical program, which is used to calculate the statistical results of this experiment. Figure 4-2 shows an abstraction of where the length values have been drawn from the MP3 frame. The source code for the program used to extract the information is listed in Appendix A.

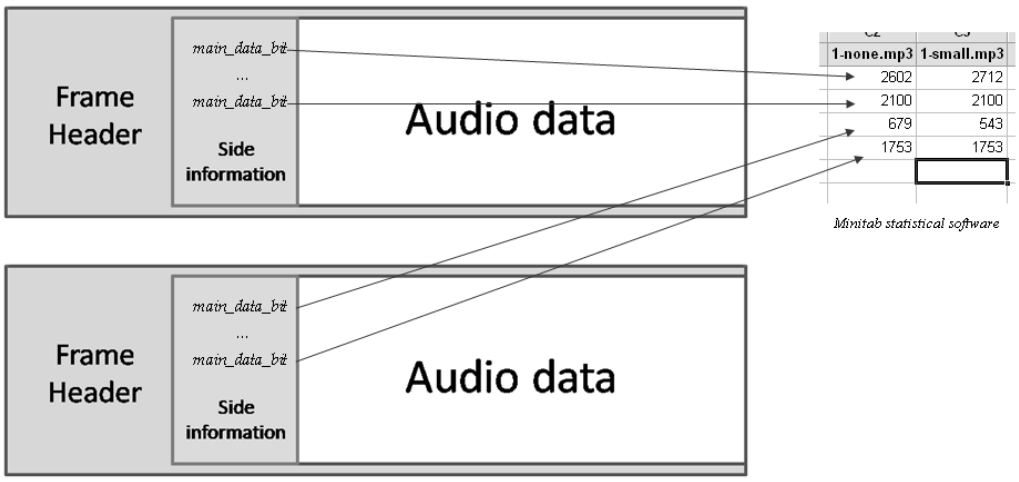


Figure 4-2: An abstraction of how the *main_data_bit* information is pulled from the side information of the MP3 frame and placed in the statistical software. Note that the data was intermediately placed in a text file before transfer to the statistical software.

We organize our samples into four groups, and also 50 sets. Our independent variable is the group of MP3 files with no embedded content; those in the column "MP3 with no embedded content" of Table 4-1. The other three columns can then be characterized as differing levels of dependent variables.

4.2 Research Hypotheses

According to the research in "Detecting low embedding rates," we should see the variances of the same source file's MP3 derivatives - with increasingly larger embedded content – increase accordingly. For our first hypothesis, we will attempt to show that the sample sets follow the trend with more than 90% of the time with a 95% confidence level. For the second hypothesis, we will examine the average variances of the sample categories as a whole. Let us define our first two hypotheses:

$H1_0$: *For each sample set the likelihood of the variance trend* $Variance_{none} \leq Variance_{small} \leq Variance_{medium} \leq Variance_{large}$ *is* $\leq .90$.

$H1_a$: *For each sample set the likelihood of the variance trend* $Variance_{none} \leq Variance_{small} \leq Variance_{medium} \leq Variance_{large}$ *is* $> .90$.

$H2_0$: *For all samples, the variance statistics follow any trend other than* $AvgVariance_{none} \leq AvgVariance_{small} \leq AvgVariance_{medium} \leq AvgVariance_{large}$

$H2_a$: *For all samples, the variance statistics follow the model* $AvgVariance_{none} \leq AvgVariance_{small} \leq AvgVariance_{medium} \leq AvgVariance_{large}$

We test this criterion on both a micro and macro level by examining the variances for the individual sample sets and by grouping all the variances by category, regardless of set.

Returning to Dr. Westfeld's work, specifically the graph depicted in Figure 3-6, we see that the graph shows a variance threshold that separates the files with embedded content from those with nothing embedded. Since this graph shows MP3 files of different sizes, and the files in our experiment are all of comparative size, it should be trivial to find this variance threshold. Let our third hypothesis be:

$H3_0$: There is no variance threshold, with respect to block length, with which to determine the existence of MP3Stego embedded content within an MP3 file.

$H3_a$: There is a variance threshold, with respect to block length, with which to determine the existence of MP3Stego embedded content within an MP3 file.

After conducting the statistical trials for these hypotheses, we will examine other statistical trends in the data in search of other predictor variables.

4. 3 Performance Measures

We gather several statistical measures on the data sets we gather. For each of the four sample groups, we gather these statistical measures. We can also use statistical software to compare these measures with respect to each of the 50 sample sets.

1) Variance

After we extracted the block lengths for each of the 200 sample files, we calculated the

variance of each of the lists of block lengths. Variance of the MP3 block lengths was calculated with the standard variance formula:

$$Variance = \sigma^2 = \frac{\sum (blocklength - \mu)^2}{n} \quad (1)$$

Where μ is the average of all block lengths in the MP3 sample file and *n* is the number of blocks in the MP3.

2) Range

The range of each data section is calculated by subtracting the smallest block length in the data set by from the largest.

3) Maximum/Minimum

The value of the maximum for a data set is found by extracting the largest block length in the set. The value of the minimum is the smallest value in the set.

While statistical testing often involves the use of tests such t-tests, in this experiment we will evaluate the performance of each statistical measure by the efficiency of the testing in discrimination between the dependent and independent variable groups – we test to see if the statistical measures reveal significant differences between MP3 files with embedded content and those that do not have embedded content.

The performance metrics used in this analysis can also be judged by the effectiveness of the statistic in classifying the sample file into the set of embedded or non-embedded files. For each statistic, the data results can be used to create a value threshold with which the sample file could be classified as either a “clean” file or a file that has been embedded with hidden content.

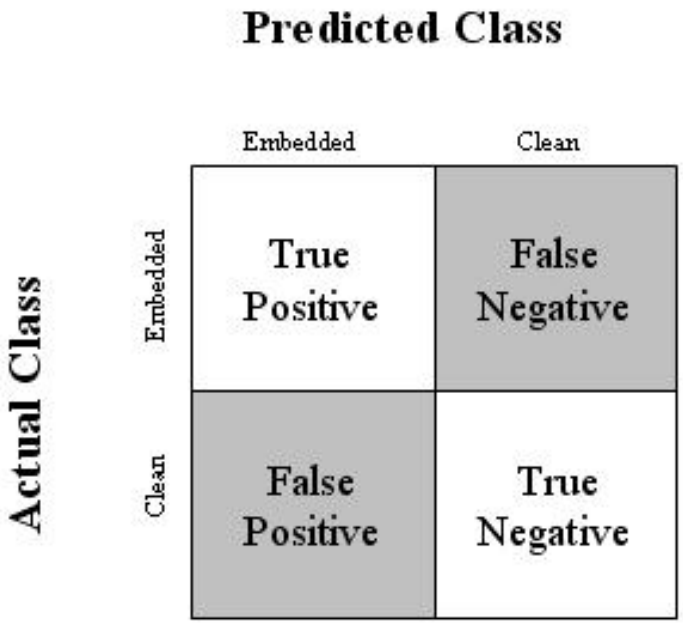


Figure 4-3: A matrix of the possible outcomes of the statistical classifiers, commonly known as a confusion matrix.

Before the performance of a given statistical predictor can be assessed, it is important to detail the different results of a classification. A predictor will decide whether a sample belongs to either the set of clean files or the set of files with embedded content. This prediction can then be either correct or incorrect. Based on these two results, the output of the prediction can be described as one of four categories: True Positive, False Positive, True Negative, or False Negative. The “True” designation is given when the assessment is correct and “False” when the prediction is incorrect. A “Positive” is assigned when the prediction assigns the sample to the set of embedded MP3 files, and “Negative” when the prediction assigns the sample to the set of clean files. In other words, the predictor succeeds if the predicted class of the sample file agrees with the actual class of the sample file.

Assigning classification results in this manner can be generalized by aligning the possible results of the actual and predicted classes in a matrix, and we have done so in Figure 4-3. Now that we have described the possible outcomes of any predictor that we use, we can discuss the meaning and implications of these results.

Of course, the most desirable outcomes are correct predictions. A True Positive result indicates that the predictor has accurately classified the sample file as a file containing MP3Stego embedded content. Conversely, the True Negative result occurs when the predictor accurately predicts a sample MP3 file is devoid of MP3Stego embedded content. Of the four possible outcomes, the True Negative and True Positive outcomes are the most desirable, but simply describing the effectiveness of a predictor in terms of success and failure is not sufficient, because of the nature of the failure cases.

A False Negative result occurs when the predictor categorizes the sample file as a clean MP3 file, but the file actually contains hidden content. The converse failure, False Positive, occurs when the predictor categorizes the sample file as having MP3Stego content but the sample file actually has no hidden content. While both of these cases are failures with respect to the predictor, there are different implications associated with these two different failures. In a realistic scenario, a False Positive can possibly place unjust implications on a cover file, entity which created or transmitted the cover file, or even transmission medium where the cover was found. In this manner, inaccurate suspicions may arise for the user of the predictor. The best case scenario for a False Positive is that user of the predictor is forced to use additional time to inspect the case in question and determine the validity of the prediction. A worse case scenario for a False Positive result is that the user of the prediction algorithm gains uncertainly in the positive results of the prediction – which is also an undesirable scenario. In contrast to the False Positive, the outcome of False Negative means that the predictor simply did not detect the hidden content. In this scenario, the predictor is dangerous because the user believes that the predictions are

accurate and has no knowledge of a steganographic object which has escaped detection – in addition to any damages the success of the hidden message may cause or enable. Because there was no detection, the prediction system cannot be reevaluated on account of failure. It is difficult to quantify the effectiveness of predictor variables because of these stipulations, and it becomes crucial to use analysis to identify how likely a predictor is to produce False Positive and False Negative results when categorizing samples.

4.4 Computational Results and Analysis

4.4.1 Variance

After the calculation was complete, we grouped each of the variances into one of the four sample groups: none, small, medium, and large. We then calculated the basic statistics of the variances of the block lengths by category. Table 4-3 shows the calculated statistics of the variances of the block lengths by category.

Figure 4-4 shows the individual variance values with respect to group. On first glance, we can see that there is much overlap between the four groups. In fact, variance does not seem to be a good predictor variable for distinguishing between files with and without hidden embedded content.

This graph does not provide a perspective on the variances of the embedded samples with respect to the same source audio MP3 encoded with no embedded content. In order to see this relationship, we need to group the variances based on sample. Figure 4-4 shows this relationship. In this graph, the *x*-axis represents the sample number, where the sample numbering scheme is sequential through the samples with respect to embedding category (e.g., 1 – 1-none.mp3, 2 – 1-small.mp3, 3 – 1-medium.mp3, 4 – 1-large.mp3, 5 – 2-none.mp3, etc.). Looking at the individual

sample groups (connected by lines) we can see that the variances with respect to the sample groups fit the trend of rising variance as more embedded content is added. This graph also shows that the initial variances for the files that do not have embedded content tend to start in the 1500 – 2500 range, but commonly are above the 5000 mark, and have a couple severe outliers.

Our analysis so far has concluded that while MP3Stego embedded content does make a statistically significant change on the variance of an encoded MP3 file, we cannot conclude that there is a certain variance limit or threshold that would enable us to classify embedded and non-embedded files. In order to see if there is a statistical measure that will, we continue our analysis by examining other statistical data.

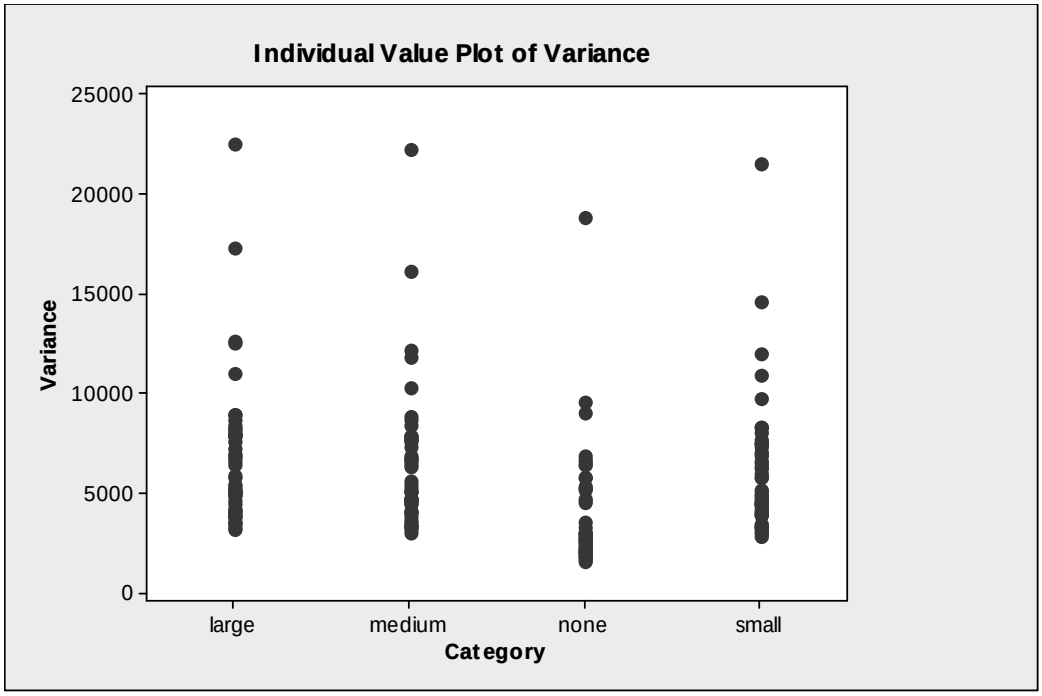


Figure 4-4: Distribution of variances, grouped by category.

Table 4-3: Statistical measures of the calculated variances of all sample files, sorted by category.

Variable	Category	Mean	SE Mean	TrMean	StDev	Variance	Minimum	Maximum
Variance	none	3912	413	3485	2918	8512708	1541	18806
	small	6085	467	5616	3305	10923505	2854	21528
	medium	6439	492	5953	3478	12097443	2972	22308
	large	6751	507	6253	3585	12849564	3148	22578

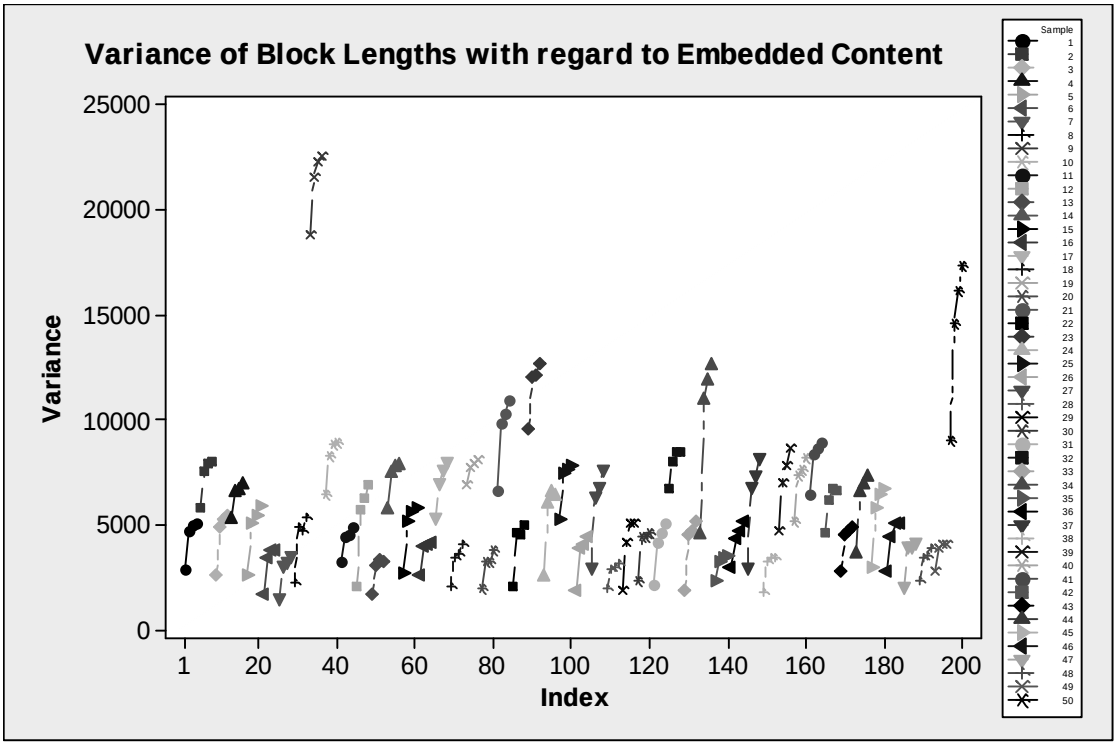


Figure 4-5: The calculated variances of the sample MP3 files, grouped according to sample set. The broken lines connect each sample set.

4.4.2 Range

Since variability is clearly an object of statistical interest in this experiment, we examine one of the most basic measures of statistical variability, the range of the block lengths. Figure 4-6 shows the individual value plot of the ranges, measured by category. We can see that the ranges

for the different categories have some overlap. Figure 4-7 shows the categorical ranges in terms of variance. In terms of range, we can see that the files with a range of less than 1500 will generally be not embedded with MP3Stego content. However, this still leaves a significant portion of our sample set that cannot be defined by this statistical predictor.

Table 4-4: Statistical measures of the calculated ranges of all sample files, sorted by category.

Variable	Category	Mean	SE Mean	TrMean	StDev	Variance	Minimum	Maximum
Range	none	1194.5	69.3	1169.9	490.2	240342.2	360.0	2469.0
	small	2436.8	64.0	2431.4	452.9	205108.1	1534.0	3322.0
	medium	2465.4	70.7	2443.7	500.0	250024.0	1592.0	3729.0
	large	2521.7	70.2	2513.5	496.5	246519.7	1593.0	3729.0

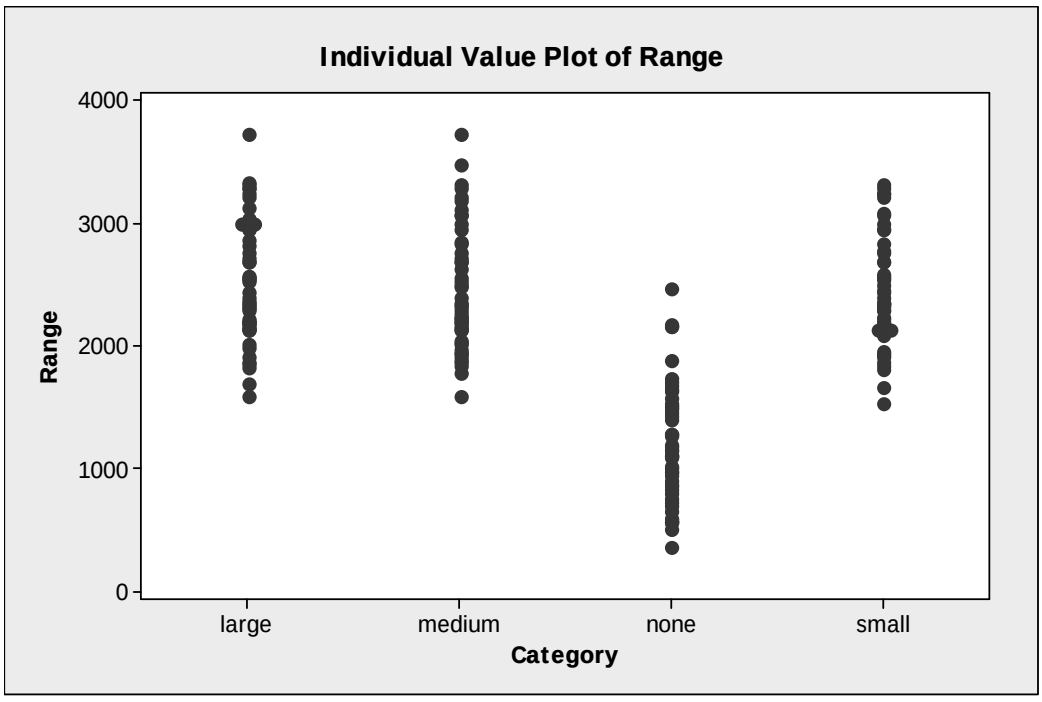


Figure 4-6: Distribution of range, grouped by category.

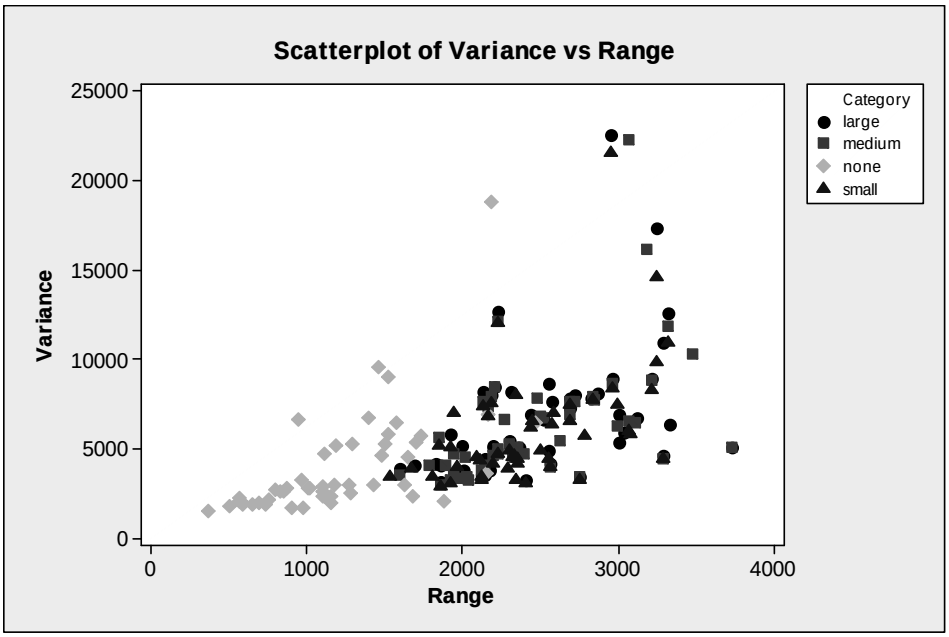


Figure 4-7: Range of the sample groups with respect to category, plotted with variance as the *y*-axis.

4.4.3 Maximum/Minimum

As the quantization process of the MP3Stego algorithm changes the values of the block lengths to accommodate the embedded content, there is the possibility that the changes in the range of the block lengths may also create significant differences in the location of the range, or more simply, change the minimum and maximum observed block length values in a way that allows the embedding process to be detected.

We examine the maximum values of the block lengths first, by category. Figure 4-8 shows the distribution of the maximum block length values by category. We can clearly see that this variable will not provide any better classification facilities than our previous variables.

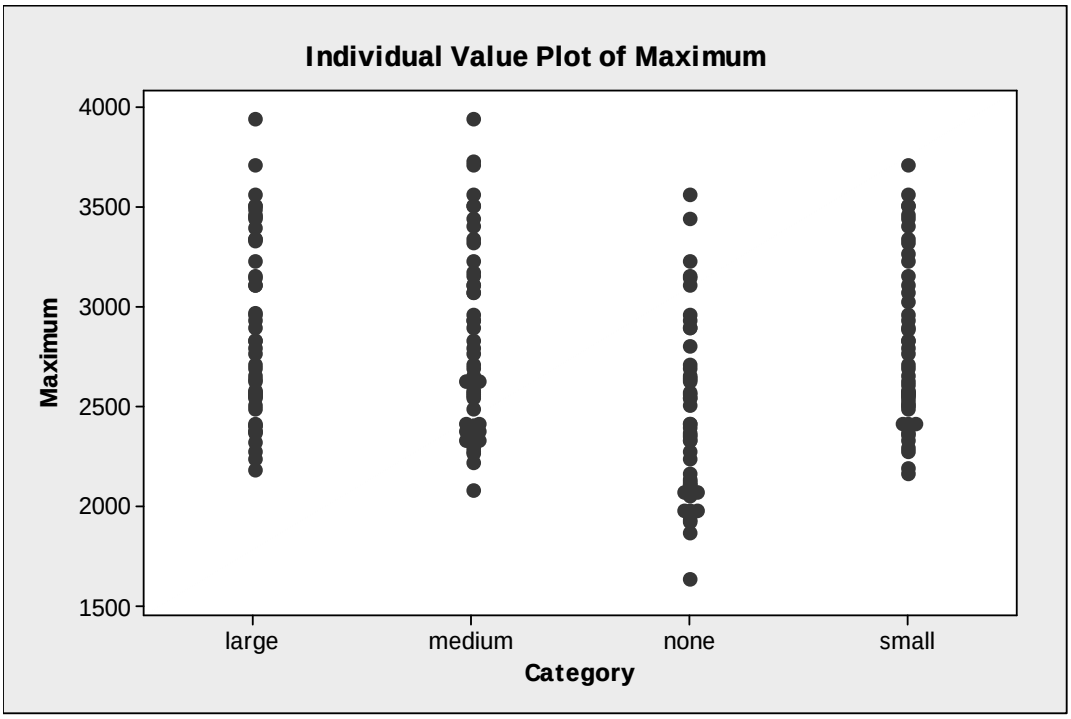


Figure 4-8: Distribution of the maximum values, sorted by category.

Table 4-5: Statistical measures of the calculated maximum values of all sample files, sorted by category.

Variable	Category	Mean	SE Mean	TrMean	StDev	Variance	Minimum	Maximum
Maximum	none	2451.4	62.1	2429.5	438.9	192589.5	1639.0	3563.0
	small	2798.4	58.2	2784.2	411.7	169.464.3	2164.0	3706.0
	medium	2820.4	66.1	2797.1	467.1	218167.4	2081.0	3942.0
	large	2877.0	63.6	2862.0	449.8	202301.0	2189.0	3942.0

Next, we examine the minimum values by category, with better results. Figure 4-9 shows the minimum block length values by category. This graph shows that there is a clearly defined relationship between embedded and non-embedded files, with only 1 sample out of the 50 “clean”

MP3 files showing a block length minimum that falls into the range of the MP3 files with hidden content.

Table 4-6: Statistical measures of the calculated minimum values of all sample files, sorted by category.

Variable	Category	Mean	SE Mean	TrMean	StDev	Variance	Minimum	Maximum
Minimum	none	1256.9	34.4	1287.9	243.2	59166.2	108.0	1431.0
	small	361.6	26.8	364.5	189.8	36042.3	0.0	701.0
	medium	355.0	25.6	357.6	180.7	32667.9	0.0	701.0
	large	355.3	25.9	359.1	183.1	33511.6	0.0	701.0

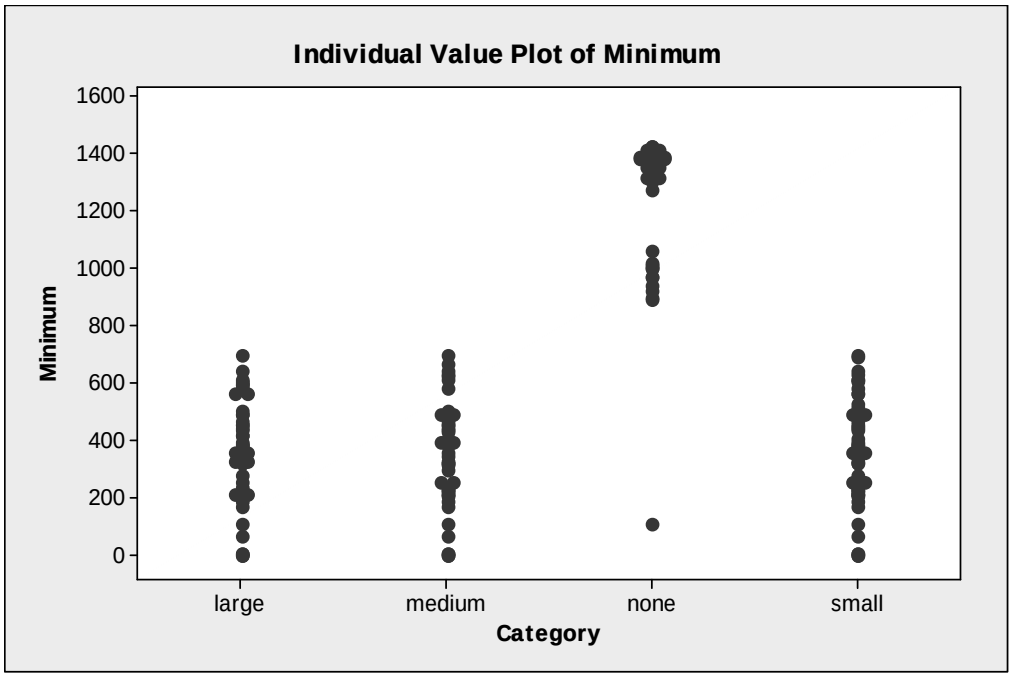


Figure 4-9: Distribution of the minimum values, sorted by category.

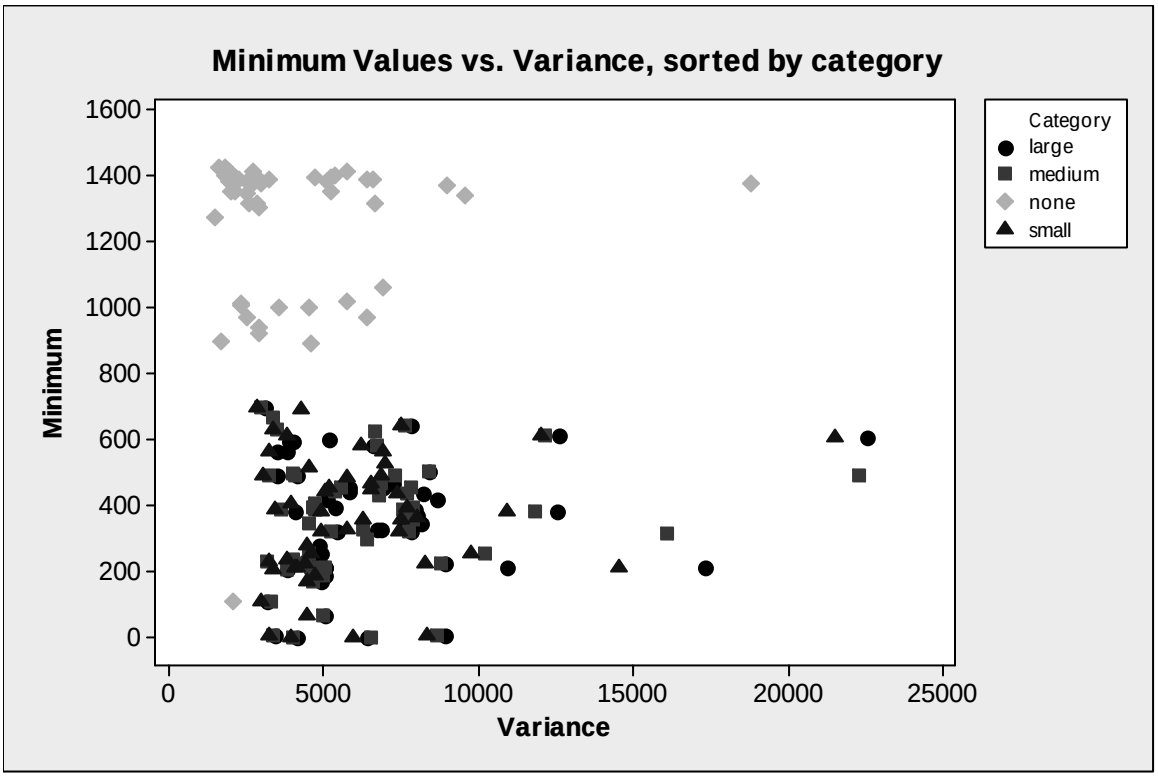


Figure 4-10: Scatterplot of minimum block length values vs. variance, sorted by sample category.

4.5 Discussion and Implications

The results of the statistical calculations provide some insight into the differences between MP3 files embedded with MP3Stego content and those which do not. However, the results show that differentiating between embedded and non-embedded files cannot be accomplished via calculating the variance of the MP3 audio block lengths.

By examining Figure 4-5, we can see that 49 out of the 50 sample sets (sample 20) in the experiment follows the fit $Variance_{none} \leq Variance_{small} \leq Variance_{medium} \leq Variance_{large}$ with respect to the samples made from one file. To understand the significance of this test, we use a one-proportion statistical test. Using Minitab statistical software, we calculate that 49 successes out of 50 trials yields a point-estimate of .98, a 95% confidence interval with a lower bound of

.9086, and testing against the proportion .9, we obtain a *p*-value of 0.034. With these results, we reject $H1_0$ at 95% confidence, and accept our alternative hypothesis: the trend for samples' variance to increase with the addition of embedded content is statistically significant.

When we examine the basic descriptive statistics displayed in Table 4-3 we can see that the averages of the variances, and therefore the variances of the categories, also follow the fit of $AvgVariance_{none} \leq AvgVariance_{small} \leq AvgVariance_{medium} \leq AvgVariance_{large}$. Because we can see that the trend follows the trend of the alternative hypothesis for H2, we set up several paired t-tests to show the statistical significance of the variances' mean relationships. Each sample category is made up of 50 sample files which are created in direct relationship to the same 50 samples in the other groups, so the paired t-test is appropriate. The three tests compare the groups "none" to "small", "small" to "medium", and "medium" to "large". Each of the paired t-tests is constructed to show statistically that the mean of the variances of the smaller group is less than the mean of the variances of the larger group. The statistical software Minitab was used to conduct the three tests. The three *p*-value results of the three tests are 0.000, 0.000, and 0.000 respectively. This allows us to conclude that $AvgVariance_{none} \leq AvgVariance_{small}$, that $AvgVariance_{small} \leq AvgVariance_{medium}$, and that $AvgVariance_{medium} \leq AvgVariance_{large}$, all with p-values of 0.000, and therefore that $AvgVariance_{none} \leq AvgVariance_{small} \leq AvgVariance_{medium} \leq AvgVariance_{large}$, which allows us to reject $H2_0$ and accept our alternative hypothesis, $H2_a$, with greater than 99% confidence. The results of these statistical tests and the Minitab output are included in the second section of Appendix B of this thesis.

The rejection of $H1_0$ and $H2_0$ has shown us that the variance of the block lengths of the MP3 files with embedded content clearly increase as more embedded content is added to the file. We then turn our attention toward determining a reliable method of segregating the populations of embedded and clean MP3 files. However, examining the distribution of variances with respect to the sample groups certainly does not allow for a threshold value of variance to segregate the

populations of embedded MP3 and clean MP3 files, as can be seen by the previous figures. We cannot say with confidence that this immediately disproves Dr. Westfeld's observations that variance cannot be used to differentiate embedded MP3 files, because the algorithm that he uses also involves the size of the MP3 file. Even though we have narrowed down the size of our sample files to the range of 3.9 – 4.2 megabytes, that still leaves some room for transformation. However, the size of the sample files is descending in nature; the largest sample file is sample 1, and the smallest is sample 50. Given this, if Westfeld's theory holds, we should see a descending trend among the variances of the MP3 sample files that are not embedded. Looking at Figure 4-5, we can see there is no such trend – in fact there are several severe outliers with regards to variance. Because of this, we have evidence that rejects Westfeld's hypothesis that MP3Stego embedded content can be detected by examining the variance of the MP3 audio blocks. Because of this, we must fail to reject $H3_0$, and conclude that there is no variance threshold with which MP3Stego embedded content can be detected.

Although the variances cannot provide a method of detection, the trend of variance is to increase as the amount of content is embedded. As Westfeld's prior research indicates, as the MP3Stego algorithm alters the size of the audio blocks, it changes the variance of the block sizes. This does not account for the outliers in the variances in the sample set. Further research into the MP3 encoding process will be needed to understand why these outliers exist.

Other statistical attempts display trends in the results. As variance is a measure in variability in the data, the range of the block length data also increases as content is embedded into the MP3 files. In Figure 4-6, we can see that the ranges of the block length data for a large section of the non-embedded files fall below the 1500 mark, and the majority of the embedded ranges are above this mark. This would allow us to classify the greater portion of embedded content.

The most definitive statistic we have found in our analysis of the audio block length data with regards to detection classification is the minimum block length size. As we can see from Figure 4-9 and Figure 4-10, there is a clear distinction between the minimum values of the block length data of MP3 files that are embedded versus those that have no embedded content. Drawing an imaginary reference line on the value 800 on this graph would segregate the populations - if we were to screen each of the 200 sample files by a minimum value threshold of 800, we would correctly identify 99.5% (199/200) of the files as embedded or non-embedded.

Table 4-7: Results of the predictor "Minimum" with the threshold value set to 800, categorized by confusion matrix.

Result	Number (%)
True Positive	150 (75%)
True Negative	49 (24.5%)
False Positive	1 (.5%)
False Negative	0 (0%)
Total Accuracy	99.5%

In order to better understand our results for this statistical predictor, we examine the predictor's results by using the confusion matrix defined in Section 4.3. Using the confusion matrix to categorize the predictions produces the results shown in Table 4-7. We see that the success percentage, denoted by results of True Positive or True Negative, comprise 99.5% of the results, making the predictor quite accurate. However, the 1 sample that was incorrectly categorized falls into the result category of False Positive, which is undesirable. Looking at the data graph in Figure 4-7, we would need to set the threshold value below 200 in order to make this classification into a True Positive, but doing so would shift the majority of the other results into the category of False Negative, which would effectively ruin the predictor's accuracy. It seems that to retain the accuracy, we must accept this False Positive. More research will be

necessary to explain why this particular sample did not fit the same pattern of minimum block length value as the rest of the sample files.

It may be noted that the value “Minimum” and “Range” are the only statistical predictors that we have examined with respect to the confusion matrix. There are several reasons for this. First and most importantly, in order to make this analysis, a threshold value must be set in order to begin the statistical classification. With the predictors variance and maximum, the values of the variances and maximums are so close together that no threshold value produces any acceptable accuracy range, and so using variance or maximum as a predictor in this fashion is unreliable and inaccurate. Examining Figure 4-4 and Figure 4-8 show the closeness of the values between the 3 sample groups that contain embedded information and those that do not.

For the predictor range, the values are not so close. However, we face a difficult problem. Given the problem of false positives and false negatives we discussed previously, it is not clear whether setting the threshold value to increase accuracy is the best course of action. Table 4-8 shows two different possibilities for setting the threshold value for the predictor “Range.” Optimizing the threshold value for the highest accuracy results in an accuracy of 95%, but also includes five False Negative results, which is problematic and compromises the integrity of the predictor. The second column in Table 4-8 shows the results of optimizing the same predictor variable for the least amount of False Negatives. In this case our accuracy is lowered only slightly, to 94.5%, but the False Negatives are reduced to one. We could then change the threshold value so that the accuracy would be slightly higher at the expense of additional False Negatives. The distribution of Range values with reference lines added to denote the two threshold values discussed here can be seen in Figure 4-11.

However, with the predictor Range, the maximum accuracy remains at 95%, and this is inferior to the accuracy of the predictor Minimum, which we can see is 99.5%. Even with the False Positive, this accuracy level is superior to all other predictors. Using a one-proportion

statistical test, with a point-estimate of 199 successful predictions out of 200 trials, we use Minitab to obtain a 95% confidence interval of (0.985225, 1.000000) using normal approximation and (0.972458, 0.999873) without using the normal approximation. Again, we could apply the normal approximation, but the low number of missed predictions challenges the assumptions of the test, so we provide the test results with and without the normal approximation in effect. This is in contrast to the 95% confidence intervals of (.919795, 0.980205) and (0.909972, 0.975766) resulting from performing the same test to the results of the Range predictor (with the threshold value optimized for higher accuracy).

In either case, the confidence interval concludes that the accuracy of using the block length minimum as a prediction variable is statistically significant. The Minitab output for these statistical tests is included in Appendix B.

Table 4-8: Possible optimizations of the "Range" predictor variable.

Result	Number (%), Optimized for accuracy (Range threshold = 1780)	Number (%), Optimized for least False Negatives (Range threshold = 1590)
True Positive	145 (72.5%)	149 (74.5%)
True Negative	45 (22.5%)	40 (20%)
False Positive	5 (2.5%)	10 (5%)
False Negative	5 (2.5%)	1 (0.5%)
Total Accuracy	95.0%	94.5%

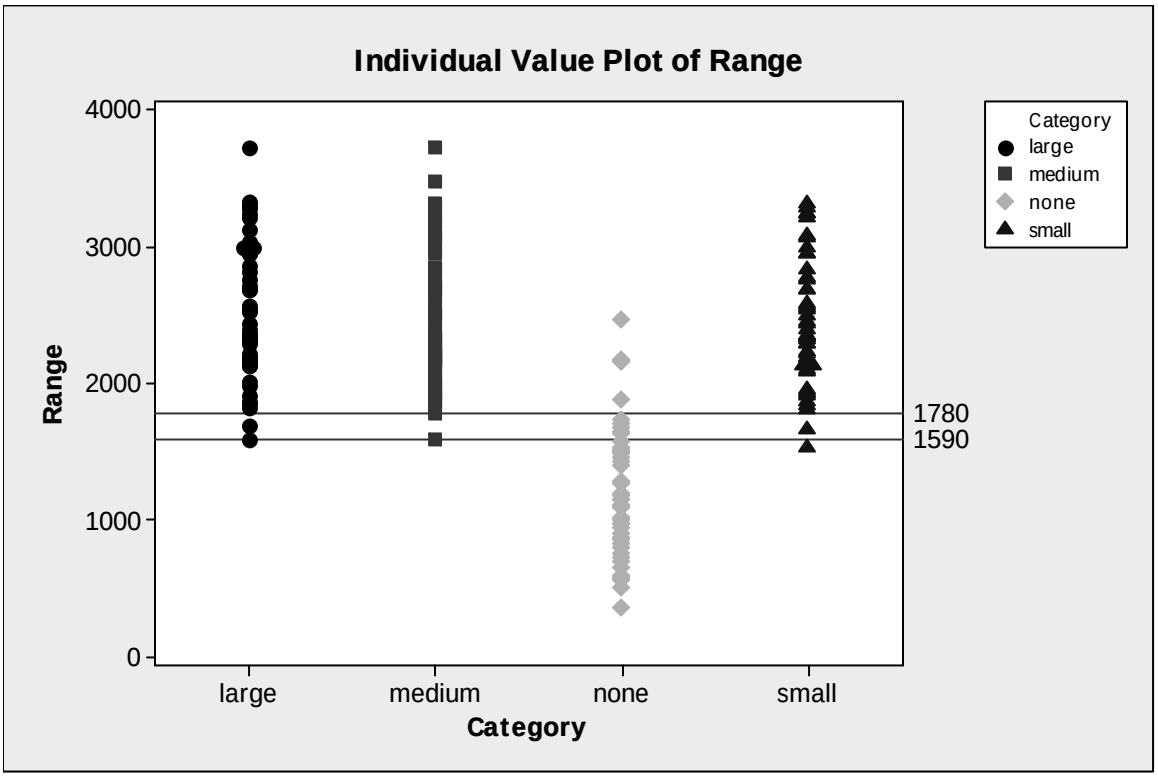


Figure 4-11: Distribution of Range values, with reference lines added to denote prediction thresholds.

Chapter 5

Conclusions

Our statistical analysis produces interesting results. From the variance testing, we conclude several results. First, we cannot conclude that variance is a substantial predictor for embedded content. Although prior research indicated that variance can be used to detect hidden content due to the increased variability of the MP3 file audio block lengths, our result data shows that the variances for the non-embedded files do not begin at a certain value, allowing block variance analysis to conclude hidden content. While the variance for both individual sample sets and the mean variance for the sample categories as a whole does increase as larger embedded content is added, which allowed us to reject two of our null hypotheses, we conclude that variance by itself is not a suitable predictor variable for reliable detection of MP3Stego embedded content. Our results also show that variance is a poor statistical predictor of MP3Stego embedded content, as a threshold value with a valid accuracy rating cannot be established from the result data – in fact, our data shows that the variances of all the sample files, embedded and clean, show a wide range of variance values.

Statistical analysis of the range variable was also indicative of change as embedded content is increased. Range can also be used as a predictor, but there is some overlap between the ranges of the embedded and non-embedded sample file ranges, so we do not conclude that range is a reliable predictor. We are able to trade a small degree of accuracy to lessen the amount of False Positives, but we still find a maximum accuracy of around 95.5%, which is inferior to the Minimum value predictor. Like variance, the maximum value of the block lengths proves to be a weak predictor variable, as the MP3Stego algorithm changes the quantization factors so that the

block lengths become smaller as the blocks are altered. Maximum shows no reliable prediction accuracy with any threshold value.

Our comparisons show that the minimum value of the block lengths is the best variable we have found to use as a predictor to classify files embedded with MP3Stego content. In our test, using the minimum as a predictor, the results were good - out of the 200 sample files, only 1 falls on the wrong side of a minimum value threshold. Although this result can be classified as a False Positive, the accuracy the minimum block length as a statistical predictor is still very good, and so we conclude that this is the strongest predictor.

Future research is necessary to conclude why the False Positive result has occurred, and why the sample file that was incorrectly categorized had a minimum block length value that was relatively far away from the minimum block length values of the rest of the sample files.

References

8hz Productions, 8hz Mp3 Encoder. Computer software, 1998.

Agaian, S.S.; Akopian, D.; Caglayan, O.; Dapos;Souza, S.A., "Lossless Adaptive Digital Audio Steganography." Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005: 903-906.

Anderson, Ross, "Stretching the Limits of Steganography." Information Hiding, 1174/1996: Springer Berlin / Heidelberg, 1996: 39-48.

Anderson, Ross J., and Fabien A. P. Petitcolas, "On the Limits of Steganography." IEEE Journal of Selected Areas in Communications, 16(4), 1998: 474-481.

Artz, Donovan, "Digital Steganography: Hiding Data within Data." IEEE Internet Computing 5(3), 2001: 75-80.

Bao, Paul, Ma, Xiaohu, "Mp3-Resistant Music Steganography Based on Dynamic Range Transform." Proceedings of 2004 International Symposium on Intelligent Signal Processing and Communications Systems, 2004. ISPACS 2004, 2004: 266-271.

Böhme, Rainer, and Westfeld, Andreas , "Statistical Characterisation of Mp3 Encoders for Steganalysis." Proceedings of the 2004 Workshop on Multimedia and Security, 2004: 25-34.

Bouvigne, Gabriel. "Personal Website". Available at: . <http://cplus.about.com/gi/dynamic/offsite.htm?zi=1/XJ/Ya&sdn=cplus&cdn=compute&tm=32&f=00&su=p284.8.150.ip_&tt=14&bt=1&bts=1&zu=http%3A//gabriel.mp3-tech.org/>. Accessed: February 24, 2009.

Cvejic, N., and Seppanen, T., "Increasing the Capacity of Lsb-Based Audio Steganography." 2002 IEEE Workshop on Multimedia Signal Processing, 2002: 336-338.

Ehmer, Richard H, “Masking by Tones Vs Noise Bands.” The Journal of the Acoustical Society of America, 31(9), 1959: 1253 - 1256.

Gang, Litao, Ali N. Akansu, and Mahalingham Ramkumar, “Mp3 Resistant Oblivious Steganography.” 2001 Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '01), 3, 2001: 1365-1368.

Gopalan, Kaliappan, Wenndt, Stanley J.; Adams, Scott F.; Haddad, Darren M., “Audio Steganography by Amplitude or Phase Modification.” Proc. SPIE - Security and Watermarking of Multimedia Contents, 5020, 2003: 67-76.

Herodotus. The Histories, Chap. 5 - the Fifth Book Entitled Terpsichore, 7 - the Seventh Book Entitled Polymnia. London, England: J. M. Dent & Sons, Ltd., 1992.

ID3 tagging standard, “Id3.Org.” Available at: http://www.id3.org/. Accessed: February 24, 2009.

Jayant, N., J. Johnston, and R. Safranek, “Signal Compression Based on Models of Human Perception.” Proceedings of the IEEE,81, 1993: 1385 - 1422.

Kelley, Jack, “Terror Groups Hide Behind Web Encryption.” USA Today, 2001, Available at: http://www.usatoday.com/tech/news/2001-02-05-binladen.htm. Accessed: February 6, 2009.

McCullagh, Declan, „Bin Laden: Steganography Master?“ wired.com, 2001. Available at: http://www.wired.com/politics/law/news/2001/02/41658. Accessed: February 6, 2009.

Noto, Mark, “Mp3stego: Hiding Text in Mp3 Files". Available at: http://www.tulane.edu/~park/courses/ElectronicMusicHistory/papers/Mp3Stego-DataHidingInMP3.pdf. Accessed: February 24, 2009.

Petitcolas, Fabien A. P, Mp3stego [Software/Web Documentation]. Computer software, 1997-2009.

Petitcolas, Fabien A. P., Anderson, Ross J., and Kahn, Markus G., “Information Hiding - a Survey.” Proceedings of the IEEE, 87(7), 1999: 1062-1078.

Platt, Christian. (2009). “UnderMP3Cover”. Computer Program. Accessed 2009. Available at: http://sourceforge.net/projects/ump3c/.

Provos, Niels, and Honeyman, Peter, “Detecting Steganographic Content on the Internet.” Technical Report CITI 01-1a (2001).

Ruckert, Martin, Understanding Mp3: Semantics, Mathematics, and Algorithms. Birkhäuser, 2005.

Schneier, Bruce, “Terrorists, Steganography, and False Alarms”. 2005. Available at: http://www.schneier.com/blog/archives/2005/08/terrorists_steg.html. Accessed: February 6, 2009.

Simmons, G. J, “The Prisoners’ Problem and the Subliminal Channel.” Workshop on Communications Security (CRYPTO '83), 1984: 51-67.

Tan, Jiaqiang, Wang, Rangding, and Yan, Diqun, “An Information Hiding Algorithm for Mp3 Audio on Vlc Domain.” 2008 Conference on Neural Networks and Signal Processing, 2008: 392-395.

Upham, Derek, “Jsteg”. Available at: http://zooid.org/~paul/crypto/jsteg/. Accessed: February 24, 2009.

Westfeld, Andreas, “Detecting Low Embedding Rates.” Information Hiding, Lecture Notes in Computer Science: Springer Berlin / Heidelberg, 2578, 2003: 324-339.

Westfeld, Andreas, “Steganalysis in the Presence of Weak Cryptography and Encoding.” Digital Watermarking, Lecture Notes in Computer Science: Springer Berlin / Heidelberg, 4283, 2006: 19-34.

Westfeld, Andreas, and Pfitzmann, Andreas, “Attacks on Stegangraphic Systems: Breaking the Steganographic Utilities Ezstego, Jsteg, Steganos, and S-Tools - and Some Lessons Learned.” Information Hiding, 1768, Springer Berlin / Heidelberg, 2000: 61-76.

Wikipedia, “Mp3”. Accessed 2009. Available at: http://en.wikipedia.org/wiki/MP3. Frame header diagram available at: http://upload.wikimedia.org/wikipedia/commons/0/01/Mp3filestructure.svg. Quantization diagram available at: *http://en.wikipedia.org/wiki/Quantization_(signal_processing)* Accessed: February 24, 2009.

Zaenuri, Achmad. “Mp3stegz”. (2009). Computer Program. Available at: http://mp3stegz.wiki.sourceforge.net/

Appendix A

Source Code for mp3_analyze.c

The program mp3_analyze.c was written in C++ and compiled on the g++ compiler on a computer running the Fedora 8 Linux operating system. The program was written with simplicity in mind; hence, bit streams are not used, rather simple get() commands to input the mp3 files' contents. The program also has the ability to extract and interpret MP3 frame header information for the user, although in this form that functionality has been commented out. In it's current form, the program simply extracts each frame's *main_data_bits* from the side information, and exports them to a text file.

```
// ------------------------------------------------
//
//    Mp3 Analyzer
//    Jonah Gregory
//
// ------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <iostream.h>
#include <fstream.h>



int get_t_layer(int index);
float get_t_version(int index);
int get_t_bitrate(int index, float version, int layer);
int get_t_samplerate(int index, float version);

int main ( int argc, char * argv[] ) {

      char * infilename = new char[48];
      memset (infilename, '\0', 48);

      if ( !argv[1] ) {
            printf( "\nEnter the name of the infile: " );
            cin >> infilename;
```

```
            } else {
                  memcpy ( infilename, argv[1], strlen(argv[1]) );
                  }

      ifstream infile (infilename, ios::binary|ios::in|ios::ate);
      if (!infile.is_open()) {
            printf("Could not open input file!\n");
            return -1;
            }

      unsigned long infile_size = 0;
      // Calculate infile size
      infile_size = (long) infile.tellg();
      infile.seekg(0, ios::beg);
      // Let's make a signpost to describe where the information tag
begins.
      unsigned long tag_loc = infile_size - 128;

      //printf("\nSize of input file: %d", infile_size);
      //printf("\nSize of input file - info tag: %d", tag_loc);

      // Need a buffer or two...
      unsigned char buf1 = 0;
      unsigned char buf2 = 0;
      unsigned char buf3 = 0;
      unsigned char buf4 = 0;
      unsigned char buf5 = 0;
      unsigned char buf6 = 0;
      unsigned char buf7 = 0;
      unsigned char buf8 = 0;
      unsigned char buf9 = 0;
      unsigned char buf10 = 0;
      unsigned char buf11 = 0;
      unsigned char buf12 = 0;
      unsigned char buf13 = 0;
      unsigned char buf14 = 0;
      unsigned char buf15 = 0;
      unsigned char buf16 = 0;
      unsigned char buf17 = 0;
      unsigned char buf18 = 0;
      unsigned long last_header_start = 0;
      unsigned long last_header_end = 0;
      int number_blocks = 0;
      int i = 0;


//    while ( (i < atoi(argv[2])) && ((int) infile.tellg() < tag_loc ))
{
      while ( (!infile.eof()) && ((int) infile.tellg() < infile_size )
) {
//    printf ("\n-------------------------------------------------\n--
--------------- Loc --> %d -----------------", (int) infile.tellg() );
      // DEBUG LINE

      number_blocks++;
```

```
// ---------------------------------------------
//    layout
// ---------------------------------------------
//   1    2    3    4    5    6    7    8    9   10   11   12   13   14
// [---][---][---][---][---][---][---][---][---][---][---][---][---][--
-]
// [------Header------][---CRC--]...
//

      buf1 = buf2 = buf3 = buf4 = buf5 = buf6 = buf7 = buf8 = buf9 = 0;
      buf10 = buf11 = buf12 = buf13 = buf14 = buf15 = buf16 = buf17 =
buf18 = 0;
      //printf("\nBufs: %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d
%d %d", buf1, buf2, buf3, buf4, buf5, buf6, buf7, buf8, buf9, buf10,
buf11, buf12, buf13, buf14, buf15, buf16, buf17, buf18);
      buf1 = infile.get();
      buf2 = infile.get();
      buf3 = infile.get();
      buf4 = infile.get();
      buf5 = infile.get();
      buf6 = infile.get();
      buf7 = infile.get();
      buf8 = infile.get();
      buf9 = infile.get();
      buf10 = infile.get();
      buf11 = infile.get();
      buf12 = infile.get();
      buf13 = infile.get();
      buf14 = infile.get();
      buf15 = infile.get();
      buf16 = infile.get();
      buf17 = infile.get();
      buf18 = infile.get();

      //printf("\nBufs: %d %d %d %d %d %d %d %d [%d %d] %d %d %d %d %d
[%d %d %d]", buf1, buf2, buf3, buf4, buf5, buf6, buf7, buf8, buf9,
buf10, buf11, buf12, buf13, buf14, buf15, buf16, buf17, buf18);
      //printf("\nBufs: %d %d %d %d %d %d %d %d %d", buf1, buf2, buf3,
buf4, buf5, buf6, buf7, buf8, buf9);
      //printf("\nBufs: %d %d %d %d", buf1, buf2, buf3, buf4);

      // These are the header-derived descriptors; denoted by 'h_' ----
------
      int h_mpeg_version = (buf2 & 24) >> 3;
      int h_layer_descript = (buf2 & 6) >> 1;
      int h_bitrate = (buf3 & 240) >> 4;
      int h_sampling_rate = (buf3 & 12) >> 2;
      int padding = (buf3 & 2) >> 1;
      int crc_bit = (buf2 & 1);
      // ---------------------------------------------------------------
------

      // Translate these into the actual numbers; denoted by 't_' -----
------
```

```
      int t_layer = get_t_layer(h_layer_descript);
      float t_mpeg_version = get_t_version(h_mpeg_version);
      int t_bitrate = get_t_bitrate(h_bitrate, t_mpeg_version,
t_layer);
      int t_sampling_rate = get_t_samplerate(h_sampling_rate,
t_mpeg_version);
      //-----------------------------------------------------------------
------

      // Frame description; pretty much for DEBUG at this point -------
------
      //printf("\nFrame %d -- ", number_blocks);
      //printf("\nMPEG Version: %1.1f (%d)", t_mpeg_version,
h_mpeg_version );
      //printf("\nLayer Descriptor: %d (%d)", t_layer, h_layer_descript
);
      //printf("\nBitrate: %d (%d)", t_bitrate, h_bitrate );
      //printf("\nSampling Rate: %d (%d)",t_sampling_rate,
h_sampling_rate );
      //printf("\nPadding Bit: %d", padding );
      //printf("\nCRC bit: %d", crc_bit);
      // -----------------------------------------------------------------
------

      // 7       8       9       10      11      12      13      14
15
      //
12345678123456781234567812345678123456781234567812345678123456781234567
8
      // [--r_s--][pad][sh][---mdb----][-------][------][--]|[---------
----][--][
      //
      // 16      17      18      19      20      21      22      23
24
      //
12345678123456781234567812345678123456781234567812345678123456781234567
8
      // -]|||[---mdb----]

      // Get reservoir size -- * TO DO (maybe) this needs to be
compensated for no CRC
      //int reservoir_size = (buf7 << 1) | ((buf8 & 128) >> 7);
      //printf("\nReservoir size: %d", reservoir_size);

      // Get the main data bits
      int mdb1 = 0, mdb2 = 0;

      // Check value of CRC bit to see where to get main data bits
      if ( crc_bit == 1 ) {
            //printf("\nNo CRC protection...");
            mdb1 = (buf7 & 63) << 6;
            mdb1 = mdb1 | ((buf8 & 252) >> 2);

            mdb2 = (buf14 & 7) << 9;
            mdb2 = mdb2 | (buf15 << 1);
```

```
            mdb2 = mdb2 | ((buf16 & 128) >> 7);

            } else {
                  //printf ("\nCRC protection found...");
                  mdb1 = (buf9 & 63);
                  mdb1 = mdb1 << 6;
                  mdb1 = mdb1 | ((buf10 & 252) >> 2);

                  mdb2 = (buf16 & 7) << 9;
                  mdb2 = mdb2 | (buf17 << 1);
                  mdb2 = mdb2 | ((buf18 & 128) >> 7);
                  }

      printf("\t%d\t%d", mdb1, mdb2);


      int frame_length = (144 * t_bitrate * 1000 / t_sampling_rate) +
padding;
      //printf("\nFrame Length: %d",frame_length );

      int offset = frame_length - 18;
      infile.seekg(offset, ios::cur);

      i++;
      }

      //printf("\n\nTotal frames found: %d", number_blocks);

      // Cleanup...
      infile.close();
      delete[] infilename;
//    printf ("\n");
      }


int get_t_layer(int index)
{
      switch (index) {
            case 0:
                  return -1;
                  break;
            case 1:
                  return 3;
                  break;
            case 2:
                  return 2;
                  break;
            case 3:
                  return 1;
                  break;
            }
}

float get_t_version(int index)
{
```

```
      switch (index) {
            case 0:
                  return 2.5;
                  break;
            case 1:
                  return -1;
                  break;
            case 2:
                  return 2;
                  break;
            case 3:
                  return 1;
                  break;
            }
}

int get_t_bitrate(int index, float version, int layer)
{
 if (version == 1) {
  switch (layer) {
            case 1:
            switch(index){
                  case 1: return 32; break;
                  case 2: return 64; break;
                  case 3: return 96; break;
                  case 4: return 128; break;
                  case 5: return 160; break;
                  case 6: return 192; break;
                  case 7: return 224; break;
                  case 8: return 256; break;
                  case 9: return 288; break;
                  case 10: return 320; break;
                  case 11: return 352; break;
                  case 12: return 384; break;
                  case 13: return 416; break;
                  case 14: return 448; break;
                  case 15: return -1; break;
                  }
            break;
            case 2:
            switch(index){
                  case 1: return 32; break;
                  case 2: return 48; break;
                  case 3: return 56; break;
                  case 4: return 64; break;
                  case 5: return 80; break;
                  case 6: return 96; break;
                  case 7: return 112; break;
                  case 8: return 128; break;
                  case 9: return 160; break;
                  case 10: return 192; break;
                  case 11: return 224; break;
                  case 12: return 256; break;
                  case 13: return 320; break;
                  case 14: return 384; break;
```

```
                case 15: return -1; break;
                }
            break;
            case 3:
            switch(index){
                case 1: return 32; break;
                case 2: return 40; break;
                case 3: return 48; break;
                case 4: return 56; break;
                case 5: return 64; break;
                case 6: return 80; break;
                case 7: return 96; break;
                case 8: return 112; break;
                case 9: return 128; break;
                case 10: return 160; break;
                case 11: return 192; break;
                case 12: return 224; break;
                case 13: return 256; break;
                case 14: return 320; break;
                case 15: return -1; break;
                }
            break;
            }
     }
else if (version >=2 ) {
 switch (layer) {
            case 1:
            switch(index){
                case 1: return 32; break;
                case 2: return 48; break;
                case 3: return 56; break;
                case 4: return 64; break;
                case 5: return 80; break;
                case 6: return 96; break;
                case 7: return 112; break;
                case 8: return 128; break;
                case 9: return 144; break;
                case 10: return 160; break;
                case 11: return 176; break;
                case 12: return 192; break;
                case 13: return 224; break;
                case 14: return 256; break;
                case 15: return -1; break;
                }
            break;
            case 2:
            switch(index){
                case 1: return 8; break;
                case 2: return 16; break;
                case 3: return 24; break;
                case 4: return 32; break;
                case 5: return 40; break;
                case 6: return 48; break;
                case 7: return 56; break;
                case 8: return 64; break;
```

```
                case 9: return 80; break;
                case 10: return 96; break;
                case 11: return 112; break;
                case 12: return 128; break;
                case 13: return 144; break;
                case 14: return 160; break;
                case 15: return -1; break;
                }
          break;
          case 3:
          switch(index){
                case 1: return 8; break;
                case 2: return 16; break;
                case 3: return 24; break;
                case 4: return 32; break;
                case 5: return 40; break;
                case 6: return 48; break;
                case 7: return 56; break;
                case 8: return 64; break;
                case 9: return 80; break;
                case 10: return 96; break;
                case 11: return 112; break;
                case 12: return 128; break;
                case 13: return 144; break;
                case 14: return 160; break;
                case 15: return -1; break;
                }
          break;
          }
    }

}

int get_t_samplerate(int index, float version)
{

    if ( version == 1 ) {
          switch (index) {
                case 0: return 44100; break;
                case 1: return 48000; break;
                case 2: return 32000; break;
                case 3: return -1; break;
                }
          }
    else if ( version == 2 ) {
          switch (index) {
                case 0: return 22050; break;
                case 1: return 24000; break;
                case 2: return 16000; break;
                case 3: return -1; break;
                }
          }
    else if ( version == 2.5 ) {
          switch (index) {
                case 0: return 11025; break;
```

```
                case 1: return 12000; break;
                case 2: return 8000; break;
                case 3: return -1; break;
                }
        }
}
```

Appendix B

Statistical Trials

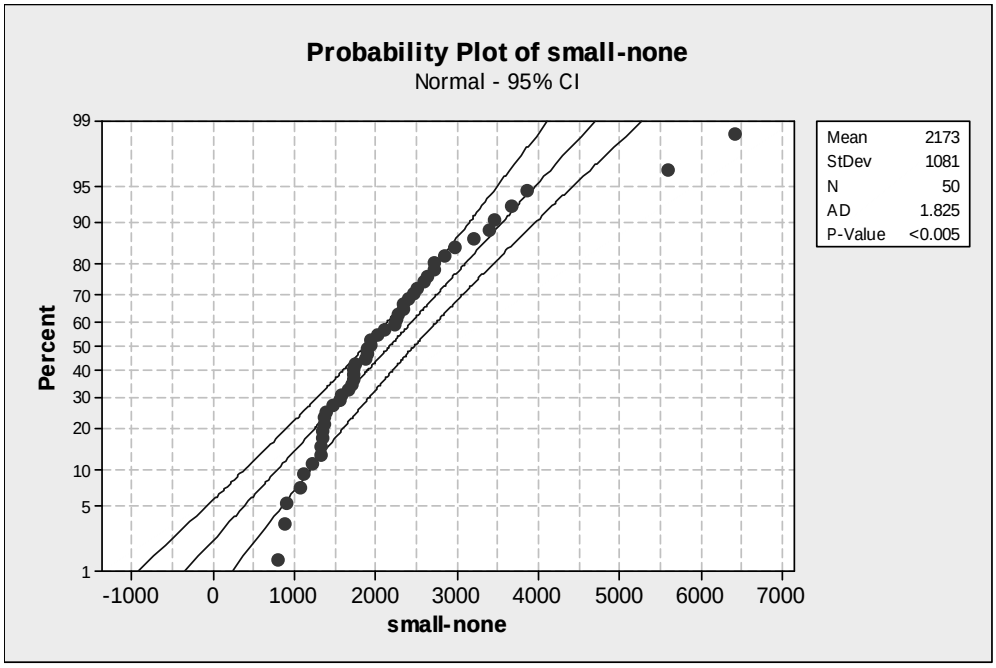
The first section of this appendix lists the statistical tests for the first hypothesis. This hypothesis tests the trend of 50 samples with 49 “successes”, and we test to see with what significance we can project our results. For this, we use a one proportion test. 49 events out of 50 trials gives us a point-estimate of .98, and at the 95% confidence level, our interval’s lower bound is .908602. This results in a *p*-value of 0.034, enough to reject our null hypothesis at the 95% confidence level.

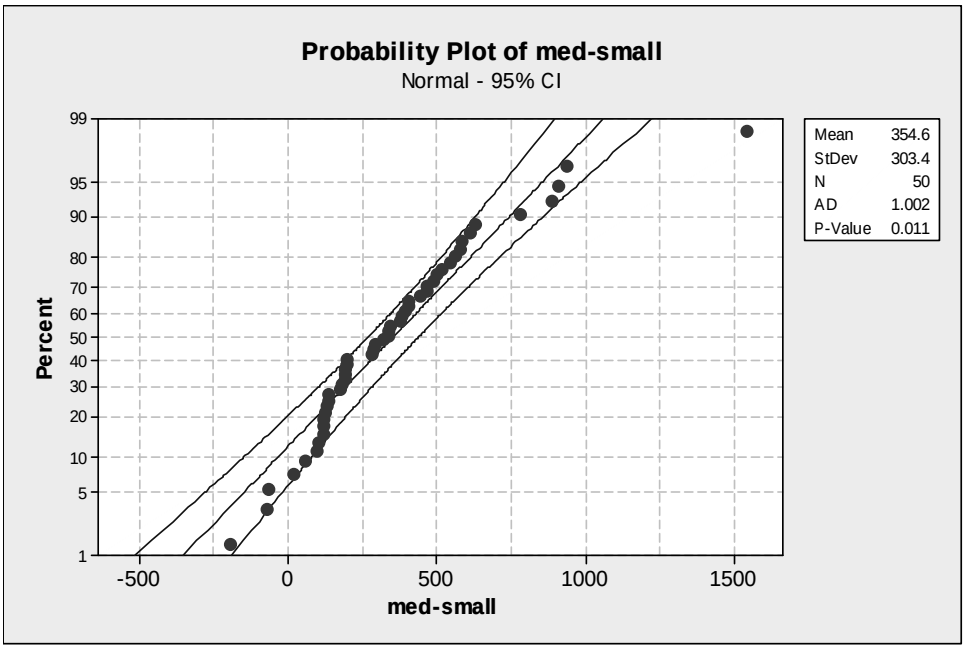
Test and CI for One Proportion

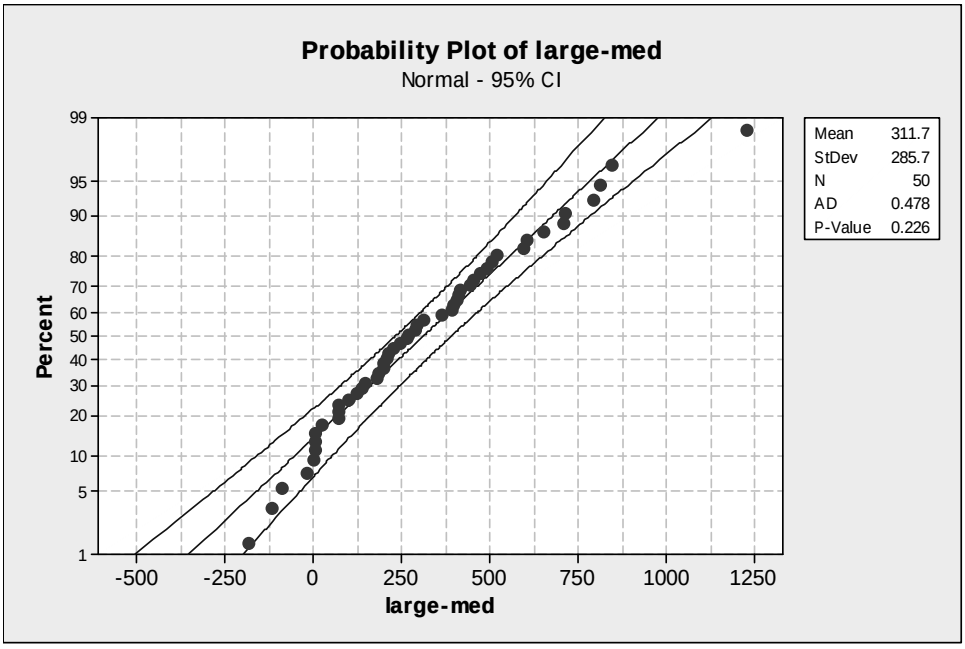
```
Test of p = 0.9 vs p > 0.9


                           95% Lower    Exact
Sample   X   N  Sample p      Bound  P-Value
1       49  50  0.980000   0.908602    0.034
```

The second section in this appendix is a listing of the Minitab output for the paired t-tests used to show with confidence the trend of the variances by category to have an ascending average (or mean). In this test, paired t-tests are appropriate because each data row references the change of one particular source file through an increasing amount of embedded content. To properly show the ascending trend, we use three paired t-tests: one between the categories “none” and “small”, a second between “small” and “medium”, and the last between “medium” and “large”.

All three t-tests show a *p*-value of 0.000 when testing the hypothesis that the first column's mean is less than the larger column's mean, allowing us to reject our null hypothesis and accept our alternative hypothesis for H2. First, the graphs displaying the normality assumption for the three tests. The first graph, the difference between the "small" and "none" variances, has several outliers but is still close to normal. The second and third graphs, the differences between "medium" and "small" and "large" and "medium", display populations that are very close to normal.







Results for: variances

Paired T-Test and CI: none, small

```
Paired T for none - small

             N   Mean  StDev  SE Mean
none        50   3912   2918      413
small       50   6085   3305      467
Difference  50  -2173   1081      153


95% upper bound for mean difference: -1916
T-Test of mean difference = 0 (vs < 0): T-Value = -14.21  P-Value = 0.000
```

Paired T-Test and CI: small, medium

```
Paired T for small - medium

             N    Mean  StDev  SE Mean
small       50    6085   3305      467
medium      50    6439   3478      492
Difference  50  -354.6  303.4     42.9


95% upper bound for mean difference: -282.6
T-Test of mean difference = 0 (vs < 0): T-Value = -8.26  P-Value = 0.000
```

Paired T-Test and CI: medium, large

```
Paired T for medium - large

             N    Mean  StDev  SE Mean
medium      50    6439   3478      492
large       50    6751   3585      507
Difference  50  -311.7  285.7     40.4


95% upper bound for mean difference: -243.9
T-Test of mean difference = 0 (vs < 0): T-Value = -7.71  P-Value = 0.000
```

The third statistical output in this appendix is the Minitab output for the statistical significance of the minimum block length as a prediction variable, and the block ranges as a prediction variable. Again, we use a one proportion test. Much like the previous test, we show the results for both using the normal approximation and without.

Test and CI for One Proportion (Minimum)

```
Sample    X    N  Sample p         95% CI
1       199  200  0.995000  (0.985225, 1.000000)

Using the normal approximation.
The normal approximation may be inaccurate for small samples.
```

Test and CI for One Proportion (Minimum)

```
Sample    X    N  Sample p         95% CI
1       199  200  0.995000  (0.972458, 0.999873)
```

Test and CI for One Proportion (Range)

```
Sample    X    N  Sample p         95% CI
1       190  200  0.950000  (0.919795, 0.980205)

Using the normal approximation.
```

Test and CI for One Proportion (Range)

```
Sample    X    N  Sample p         95% CI
1       190  200  0.950000  (0.909972, 0.975766)
```