JPEG STEGANALYSIS & TCP/IP STEGANOGRAPHY

BY

NEIL R. BENNETT

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE & STATISTICS

UNIVERSITY OF RHODE ISLAND

2009

MASTER OF SCIENCE THESIS

OF

NEIL R. BENNETT

APPROVED:

Thesis Committee:

Major Professor ______________________________

____

____

____

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2009

ABSTRACT

This thesis presents two areas of research connected steganography. In the first section we focus on detection of steganography in JPEGs. Using current detection methods proposed by Fridrich et al we examine the effect of a real-world implementation of the detection scheme in order to facilitate usage of a detection tool for law enforcement. Fridrichs work is based around machine learning and as such models must be trained on a given data set. In order for a tool to be used by law enforcement we must consider various differences of JPEG's that could be encountered during an investigation. We examine the effects of JPEG quality changes and levels of data embedding to see what models need to be trained on.

In the second part of this work we present a novel technique for communication of data across a TCP network. This is achieved by purposely modifying the TCP checksum of a packet so that the packet appears corrupt. This technique allows two machines to communicate across a routed network as no network routing device will examine the TCP checksum. This research is important to law enforcement organizations as it shows the potential for covert communications.

ACKNOWLEDGMENTS

I would like to thank my Major Professor and advisor Prof. Vic Fay-Wolfe who has been instrumental in my graduate career. His support in both encouraging my academic endevours as well as providing financial support have been critical to my work.

I would also like to thank Prof. James Kowalski and Prof. Yan Sun for readily being part of my commitee and the help and assistance they have provided.

Further thanks must be given to my fellow students and co-workers. In particular Kevin Bryan and Raghu Menon for allowing me to bounce ideas off them, provide help and advice, and pushing me intelectually. In addition I'd like to thank Krissy, Britnee and Jake in the Digital Forensics Center for letting me distract them so frequently.

Finally I would like to thank my family for their support during the time I have been at URI especially during some trying moments.

PREFACE

This thesis is presented in manuscript format with two manuscripts on the topic of steganography.

In the first manuscript we investigate model construction for steganography detection by testing variations to assumptions that have been made in academic research thus far. We do this in order to produce a set of guidelines that can be used to construct models that can be used by law enforcement in practice.

The second manuscript presents a novel technique to communicate covertly using steganography across a TCP/IP network. The technique makes packets containing hidden data appear as corrupt packets. This allows data to be sent which will then be dropped as per usual network communication between the hosts. However, this data will be extracted from the corrupt packet by the receiving host before the packet is dropped. If the packet was intercepted then it would merely appear as a network error. This algorithm is presented to examine the feasibility of it as a steganographic technique. This will bring to this type of technique to the attention of law enforcement officers who may be required to intercept network traffic and interpret it to see if it contains steganography.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iii

PREFACE iv

TABLE OF CONTENTS v

LIST OF TABLES viii

LIST OF FIGURES ix

MANUSCRIPT

1 JPEG Steganalysis 1

1.1 Abstract 1

1.2 Introduction 2

1.2.1 JPEG Compression 3

1.2.2 Steganography in JPEG 6

1.2.3 Steganalysis of JPEGs 8

1.3 Methodology 10

1.3.1 JPEG Quality 12

1.3.2 Embedding Percentage 13

1.4 Results 14

1.4.1 Baseline Models 14

1.4.2 JPEG Quality Variations 15

1.4.3 Embedding Percentage Variations 19

Page

1.5 Evaluation 20
1.5.1 JPEG Quality Variation 21
1.5.2 Embedding Percentage Variation 23
1.6 Conclusion 26
List of References 28
2 TCP Steganography 29
2.1 Abstract 29
2.2 Introduction 29
2.3 TCP/IP 30
2.3.1 TCP Header 31
2.3.2 TCP Connection Establishment 32
2.3.3 Reliable Delivery of TCP packets 34
2.4 TCP/IP Steganography 36
2.5 Hiding data in corrupt TCP packets 37
2.5.1 Sender 39
2.5.2 Receiver 42
2.6 Implementation 43
2.7 Evaluation 45
2.7.1 Imperceptibility 45
2.7.2 Undetectability 46
2.7.3 High Capacity 48
2.7.4 Robustness 48
2.8 Future Work 50

Page

2.9 Conclusion 51

List of References 52

APPENDIX

TCP Steganography Tool Source Code 53

A.1 TCPStegSend.c 53

A.2 TCPStegRecv.c 57

BIBLIOGRAPHY 63

LIST OF TABLES

Table **Page**

1 Baseline Model 95% Confidence Interval Accuracies for 75 quality . 14

2 Baseline accuracy for 55, 65, 85 and 95 quality levels at 25% embedding 15

3 Quality Variance against a 75 quality 25% embedding model 15

4 Quality Variance against a 55 quality 25% embedding model 16

5 Quality Variance against a 95 quality 25% embedding model 17

6 Testing of quality range 70 - 79 model 18

7 Matrix of Embedding Variants against single embedding model . . . 19

LIST OF FIGURES

Figure **Page**

1 JPEG Encoder Steps 3

2 Zig-zag sequence of quantized coefficients 6

3 Decision surface for blind detection of steganography 8

4 Quality variations on trained model versus baseline accuracy 17

5 Graph of accuracy for 5% embedding model when on other embeddings (solid line) versus baseline performance (dashed line) for single embedding models 24

6 TCP packet specification 33

7 TCP Connection Establishment 34

8 TCP Reliable Delivery with lost packet 36

9 TCP connection between two hosts showing how a corrupt packet can be injected into the normal stream of communication 38

MANUSCRIPT 1

JPEG Steganalysis

1.1 Abstract

Steganography is the technique of hiding information within a medium such that the data can be extracted by another party but its presence cannot be detected by human abilities. A common steganography application is within JPEG images and many algorithms exist to facilitate data hiding within them. This presents a problem for law enforcement agencies as it makes finding and presenting evidence about criminal activities complex and time consuming should steganography be used.

Steganalysis is the science of detecting steganography content and research has taken place to detect JPEG steganographic algorithms. In order to make the task of law enforcement easier it is desirable that a tool be created for fast and accurate use by them to detect steganography. Current academic research has looked at machine learning techniques with sophisticated feature extraction techniques to detect JPEG steganography. However, the assumptions made in the construction of machine learning models aren't suitable for real-world use.

In this paper we investigate two common variations that would be present in any real-world use of steganography. These variations are JPEG quality and the amount of data hidden by the steganographic algorithm. We investigate the effects of these variations against assumed JPEG quality and data embedding percentage and then present a series of guidelines and techniques in order to facilitate model construction by software companies or law enforcement agencies such that a steganography detection tool could be used in practice.

1.2 Introduction

Steganography is the science of hiding information without a perceivable difference to a human viewer[1]. It is a technique that dates back to the 1600's with Bacon's cipher which hid a message within the presentation of the text[2]. Steganography consists of a cover medium and data to be embedded. A classic example of steganography is forming a message by taking the first letter of every paragraph. If paragraphs are carefully constructed in a way that the readability is not effected then the message will not arouse suspicion even though the message is essentially in "plain view". In the modern digital age, steganography has become relevant in various forms of digital media such as images and audio. By taking advantage of lossy compression algorithms we now see steganography existing in JPEG (Joint Photographic Experts Group) images as well as MP3 (MPEG Layer 3) audio[3]. The nature of these lossy compression techniques readily allows data to be embedded such that the medium is not altered in a perceptible way.

Due to steganography allowing two parties to communicate in a discrete manner it poses a problem for law enforcement organizations. Steganography allows parties to exchange information in a manner that appears to be innocent. Such activity may include terrorism plots, communication amongst child porn rings, and other criminal activity[1]. These activities are clearly of interest to law enforcement and as such steganography poses a great threat to the detection and prosecution of such individuals. It is easy to imagine that computers could contain large amounts of criminal activity data but be hidden via steganography. Therefore, steganography causes additional complexity on machine seizure investigations because evidence may be contained within otherwise innocent files which appear to human investigators as perfectly normal files. It is due to this inherent danger that law enforcement requires a tool that allows automated scans of media

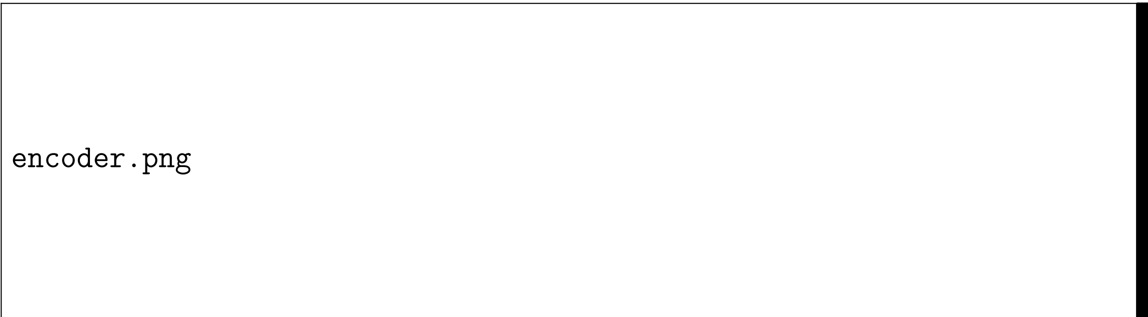

Figure 1. JPEG Encoder Steps

files that can inform them whether the file contains hidden data so that it may be investigated further in order to retrieve the evidence hidden within. Such a tool would be able to detect steganography in ways that a human investigator could not and allows processing of suspect images and audio in a much more efficient manner.

We will begin our discussion on JPEG steganography by first giving background on the JPEG compression technique. This will be followed by discussing how steganography can be achieved within JPEGs, and then finally looking at how JPEG steganography can be detected.

1.2.1 JPEG Compression

The JPEG standard was approved by the ISO (International Organization for Standardization) in 1994[4] and specifies the codec to compress and decompress an image into a stream of bytes. The JPEG standard is a lossy compression technique which means that some quality of the image is lost during the compression process. A JPEG encoder follows the pattern as shown in Figure 1 and the compression/decompression process has the following components[1]:

[1] JPEG has four modes of operation. We describe the Baseline mode, although all modes follow the same general approach

1. **DCT**

   The first part to JPEG compression is DCT (Direct Cosine Transform) compression. The DCT takes the image from the spatial domain to the frequency domain. The advantage of using the DCT is that it concentrates the "energy" of the image so that latter parts of the process can achieve higher compression ratios. The DCT is performed on the image by broking it up into a series of 8x8 blocks, consisting of the pixel values from the spatial domain, which will be input to the FDCT (Forward DCT) encoder. The 8x8 samples are then shifted from unsigned integers in the range $[0, 2^P - 1]$ to signed integers with range $[-2^{P-1}, 2^{P-1} - 1]$[5]. The FDCT is then applied to the shifted samples and the output is then an 8x8 block of samples in the frequency domain. The FDCT is performed by using the equation[5]:

   $$F_{u,v} = \frac{1}{4}C(u)C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) * cos\frac{(2x+1)u\pi}{16} cos\frac{(2y+1)v\pi}{16}\right]$$

   where $C(u), C(v) = \frac{1}{\sqrt{2}}$ for $u, v = 0$ or $C(u), C(v) = 1$ otherwise, and u, v are horizontal and vertical indices into the 8x8 block. This block now consists of a DC coefficient and 63 AC coefficients. The purpose of the FDCT is to concentrate most of the signal in the DC coefficient which is the key to achieving data compression[5]. It is important to note that the FDCT is lossless and as such has an inverse function, the IDCT, which is used during decompression of a JPEG image.

2. **Quantization**

   After the FDCT is completed each of the 64 coefficients are quantized according to a 64 element quantization table which is supplied as an input to the encoder[5]. The JPEG standard allows there to be four quantization tables that can be selected from[4]. The quantization table consists of quantizer step sizes which are used to divide a corresponding DCT coefficient followed

by rounding to the nearest integer[5]. Due to the rounding process this part is lossy. The larger the quantizer step size then the more data is lost in this process but allows for greater compression ratios. This is due to the fact that when we increase the quantization step size we force more DCT values to become zero. By adjusting the quantization table we can control the quality level of the JPEG. For example, a quantization table which all values set to 1 would correspond to a JPEG quality of 100 as no data is lost during the quantization step.

3. **Entropy Coding**

   The coding process differs between the DC and AC coefficients. For the DC coefficient, because there is correlation between adjacent DC coefficients[5], it is coded as the difference from the DC coefficient in the previous block[4]. For the AC coefficients either Huffman or Arithmetic coding can be used[2]. Quantized coefficients are traversed for entropy coding in a zig-zag manner. Figure 2 shows the path followed during entropy coding. For Huffman coding the AC coefficients are first transformed into a run-length encoding by specifying the number of zero AC coefficients before the nonzero AC coefficient, and then Huffman coding is performed on this along with a Huffman table that is supplied into the encoder[5].

4. **Encoding complete**

   At this point the 8x8 block has been fully encoded and can be outputted as a Minimum Coded Unit (MCU).

The decompression process is performed by reversing the direction of the compression process and using the quantization table to multiply out the quantized values

[2]Huffman coding is typically used due to the patents on Arithmetic coding

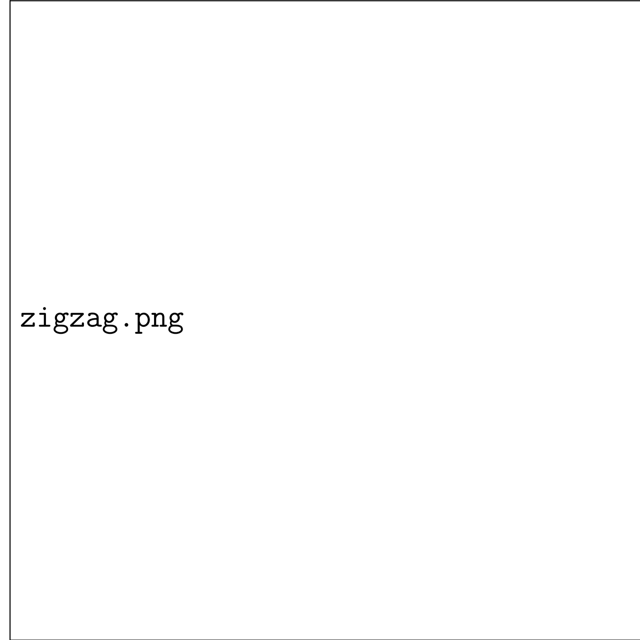

Figure 2. Zig-zag sequence of quantized coefficients

along with the IDCT function which is:

$$f_{x,y} = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v) * cos\frac{(2x+1)u\pi}{16}cos\frac{(2y+1)v\pi}{16}\right]$$

1.2.2 Steganography in JPEG

In non-compressed images steganography typically occurs in the spatial domain. Data is typically hidden in bitmap images by modifying the least significant bit (LSB) of a pixel value. This introduces a very small change in the color of the pixel that is not noticeable to the human eye[1]. By flipping the LSB values in a bitmap we can embed a binary message that can be retrieved at a later point. In the compressed JPEG image we cannot modify pixel values in the spatial domain because the JPEG compression algorithm is lossy. This means that should we try embedding data on the LSB of pixel values we may not get the same pixel value after decompression. Because of this problem steganography needs to take place in a different manner than in non-compressed bitmap images.

To embed binary data into a JPEG we embed data after the quantization step. The reason we do this is because the entropy coding that occurs after this point is lossless. Therefore any modifications we make to the 8x8 block of quantized values will remain after performing the inverse of the entropy coding. After quantization has completed there is an 8x8 block of quantized coefficients and the data embedding can take place by using LSB techniques on the quantized DCT values. Embedding data would typically not take place in the DC coefficient but rather in one of the 63 AC coefficients. The reason for this is that the DC coefficient contains most of the "energy" from the DCT transform, and therefore modifying the DC coefficient is much more likely to be visible to the human eye as a change effects more of the 8x8 block when it is decompressed. In addition it is not desirable to embed data in AC coefficients that are zero. The reasoning again is that doing so could turn the AC coefficient to a value greater than zero and as such could cause a perceptible change. Therefore the capacity of embedding data in to an image is based on the number of non-zero AC coefficients that the image contains[6]. The maximum capacity of hidden data is then closely related to both the quality level given to the JPEG as well as the contents of the image itself.

Programs that work on LSB of quantized DCT values include F5[7], Outguess[8], StegHide[9] and MBSteg[10]. These algorithms also use different techniques to select the non-zero AC coefficient to embed in. The reason for doing so is to prevent detection. If the data is embedded in a liner fashion among non-zero AC coefficients then detection becomes much more straight forward. In addition some algorithms also purposely change other AC coefficients in the image so that other second-order statistics, such as covariance and correlation between AC coefficients, are preserved which will also make detection more complex[6].

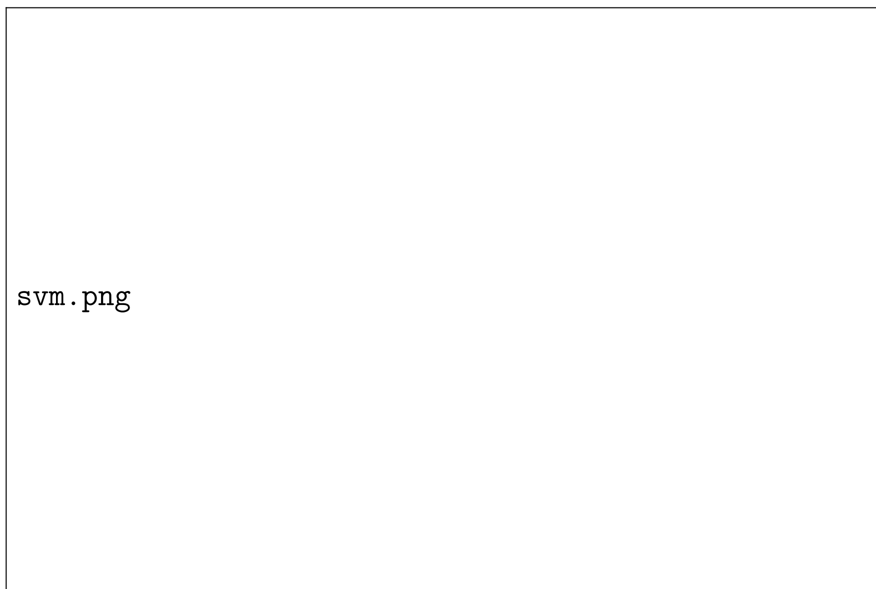

Figure 3. Decision surface for blind detection of steganography

1.2.3 Steganalysis of JPEGs

Steganalysis is the science of detecting hidden data within the given medium. Steganalysis has been effectively developed to detect LSB steganography in bitmap images such that the detection becomes deterministic. JPEG steganography presents a challenge in detecting it given the effective ways of JPEG steganographic algorithms to hide traces of embedding having taken place. Because of this there currently does not exist any deterministic algorithms for blind detection of steganography. Blind detection is when the cover image is not available. Therefore detection becomes a question of whether the image in question contains steganography or not.

Research has mainly focused around the use of non-deterministic methods to provide a reliable, if not guaranteed, prediction of whether the image contains hidden data or not. The main approach used thus far has been the use of machine learning[11] techniques. The machine learning techniques used in current research consists of supervised learning methods. These include Support Vector Machines

(SVM), Naive Bayes Classifier (NBC), and Linear Discriminant Analysis (LDA). Supervised learning means that the inputs to the machine learning model consist of labeled sets that represent the modeled environment. For the purposes of detecting steganography we develop inputs sets that consist of "clean" and "steg" images respectively. The sets themselves consist of a vector of features that are extracted from each member of the image set. The machine learning method then undergoes a "training" step wherein it uses its particular learning algorithm to be able to draw a decision line between each set. Figure 3 shows a graphical representation of a decision surface for steganography detection. After the training has taken place a model of the universe exists and additional feature vectors can be inputted to the model and a decision made as to what class they belong to. For our use in detecting steganography we are able to use this model to perform blind detection by giving it a set of features extracted from a given image and letting the machine learning technique classify it for us - informing us as to whether it likely contains steganography or not.

Clearly machine learning techniques are highly dependent on the vector of features they are given during the training procedure. Steganalysis research has mainly focused on adding, or improving, the individual features that are extracted from each image. For JPEG steganography we require a set of features that can allow a machine learning technique to determine whether the image contains hidden data or not. The defining work in this field was undertaken by Fridrich when she introduced a number of features to detect numerous steganography algorithms using 1st and 2nd order statistics on the DCT coefficients of the image[6]. Further on Shi et al developed an set of features by using a Markov process on the 8x8 coefficient blocks[12]. Pevny and Fridrich then merged these Markov based features with their own to create a feature set of 274 features[13].

In the work performed by Fridrich in [6, 13] the models created were based around an assumption of the quality level of the JPEGs used to train the model, and the percentage of data embedded into the image. The problem with these assumptions are that they do not hold for real world usage. JPEG quality level is likely to range greatly on images found on a seized computer, and furthermore there is no way of accurately telling what percentage of data might be embedded in a suspect image. For a JPEG steganography tool to be used by law enforcement we must test to see how assumptions made about the training set compare to the differences that might be found in real world usage. This research will be critical in providing a tool to law enforcement for accurate and efficient detection of steganography.

1.3 Methodology

In this section we present our methodology for testing both differences in JPEG quality and steganographic embedding percentage. First we present the overall testing methodology that is common to both differences.

In order for us to test how real world conditions would effect steganography detection while using machine learning techniques we construct models under certain assumptions and then test these against the real world differences. Because we are using machine learning techniques we must first construct the various feature vectors that will be used to train. Our training set consists of 3800 images that were obtained in from a local photographer in Nikon RAW format (NEF file format). From these RAW mode images we could then process the images to singly-compressed JPEGs of any quality level. This is important as the features extracted are dependent on the image being a singly-compressed JPEG[13]. From the 3800 images we constructed our training and testing sets with the following breakdown:

- 1400 Clean Training Images
- 1400 Steg Training Images
- 500 Clean Test Images
- 500 Steg Test Images

We are able to construct any training and testing set as needed using this breakdown. The features extracted from this training and testing set were the Merged Markov feature set as proposed by Pevny and Fridrich[13]. The feature extraction program was written by the Steganography Research Group in the URI Computer Science & Statistics group in C++ for speed reasons and was verified against the MATLAB code kindly provided by Jessica Fridrich. The machine learning technique used was LDA. Again, this was selected for speed reasons as the time required to train and predict is considerably less than the SVM learning technique and provides comparable accuracy. The key reason for working with efficient implementations is so a tool also using this can be used in an efficient manner by law enforcement. We used the steganographic algorithm F5 to provide all the steganographic images as it represents an algorithm that is readily available for end use and because all JPEG steganographic tools work on the same domain the results will be comparable for any tool.

We obtained baseline results for F5 on 75% quality JPEGs embedded with 5%, 15%, 25% and 50% maximum capacities[3]. Bootstrapping was then applied to these baseline models. Bootstrapping allows a confidence interval for the model to be obtained by testing the model with real world bias. We performed bootstrapping by forming a bootstrapping test set which is composed of the 2800 training set images along with the 1000 test set images. We then formed 200 buckets in which

[3] We used the capacity as listed by the F5 algorithm which is smaller than the capacity obtained by counting the non-zero AC coefficients

3800 images were selected with replacement. Selecting images with replacement introduces a bias (towards either clean or steg) that real world data would have. The model is then tested on each bucket and the we take the 2.5^{th} and 97.5^{th} percentiles as a range in order to get a 95% confidence error interval. This allows us to be 95% sure of the accuracy of the model in real world usage. The results will show the overall accuracy of the model along with both the false positive and false negative percentages.

We now look at the methodology for testing the quality and embedding differences.

1.3.1 JPEG Quality

Given the baseline model that we produced at 75% quality and 25% embedding we can now test this model against both clean and steg JPEGs that are of different qualities. To do this we create JPEGs in the same manner as listed above but compressed them with quality levels 55, 65, 85 and 95%. This will show the effect on using a model trained on a fixed quality then being used as for classification on JPEGs of a different quality. To test the effect of using different qualities on the 75 quality model we use the same bootstrapping approach as in section 1.3 but the bootstrap set will be composed of the clean and steg images at the different quality level. This will give us accuracy ranges for each of the 55, 65, 85, and 95 quality levels against the 75 quality model. In order to understand the accuracy variations we also provide bootstrapped evaluations of models trained on 55, 65, 85, and 95 quality all at 25% embedding. Testing a quality in this manner will allow us to plot the accuracy differences in order to identify an overall trend. We will also perform the same tests but use the 55 and 95 quality sets as training sets respectively and then test these in the same manner as we did above. Doing this will show if there is a trend for better accuracy on images below the training level

quality or above it.

After this we test a model that is trained on a range of qualities to see if the accuracy of a varied quality model is acceptable. This is performed by creating steg and non-steg images with a random quality within a given range, for example 70-79, and then testing it in two ways. First, we test it by performing bootstrapping on it which will give the overall accuracy of the model. This will show us if training on a set that has a mixed range of qualities has similar accuracy to the single quality models. Secondly we will test the model with images from just a single quality that is within the range trained on. This test will show whether the quality range model can detect a set of single quality images to the same, or similar, accuracy of the same single quality model. We will train on the range of qualities from 70-79 at 25% embedding and test on the single 75 quality 25% embedding images. Together the range tests will allow us to see if mixed quality training sets are suitable to use and to also suggest an acceptable boundary on quality ranges for detection before overall accuracy suffers.

1.3.2 Embedding Percentage

We test the variation of embedding against a model trained on a fixed embedding percentage by creating a cross matrix of models and results. This means that for each embedding percentage (5, 15, 25, 50) we will train on each one and then test the model against the remaining embedding percentages. This will create a matrix of results showing the effects of any model trained at those levels and the accuracy obtained from them when tested with images that have lower and higher embedding.

Because we are only concerned with the accuracy of detecting steganography embedding levels we do not use the bootstrapping method when generating results. Our results are obtained purely from testing the model against the set of 1400

Table 1. Baseline Model 95% Confidence Interval Accuracies for 75 quality

Embedding %	Overall Accuracy %	False Positive %	False Negative %
5	92.95 - 94.50	4.59 - 6.50	6.14 - 8.11
15	99.58 - 99.89	0.11 - 0.68	0.00 - 0.26
25	99.87 - 100	0.00 - 0.27	0.00 - 0.00
50	100 - 100	0.00 - 0.00	0.00 - 0.00

images containing steganography. This will show whether the model trained at a certain embedding percentage can detect embedding percentages both less than and greater than itself. The accuracies from these tests will show whether we need to train models specifically for embedding percentage or we can use fewer models to detect many embedding percentages.

1.4 Results

In this section we present the various results from the tests performed.

1.4.1 Baseline Models

The results of the baseline testing are given in Table 1. It is clear that the model works extremely well on embedding percentages as low as 15% with an overall error rate of less than 1%. The 5% model shows the difficulty of detecting such low embedding rates. This is a limitation of the machine learning technique as the small amount of data embedded in the image creates very little disturbance to the original images statistics. This is noticeable primarily in the increased false negative error rate. In the 15, 25 and 50% models false positives are more common but for the 5% model we observe a higher false negative rate. This is because the margin between the statistics of a steganographic image and a clean image is very small so we see more steganographic images classified as clean images.

Table 2. Baseline accuracy for 55, 65, 85 and 95 quality levels at 25% embedding

Quality	Overall Accuracy %	False Positive %	False Negative %
55	99.71 - 99.95	0.00 - 0.32	0.00 - 0.36
65	99.82 - 100	0.00 - 0.38	0.00 - 0.00
85	99.92 - 100	0.00 - 0.16	0.00 - 0.00
95	100 - 100	0.00 - 0.00	0.00 - 0.00

Table 3. Quality Variance against a 75 quality 25% embedding model

Quality	Overall Accuracy %	False Positive %	False Negative %
55	98.39 - 99.91	1.46 - 2.55	0.16 - 0.71
65	99.61 - 99.90	0.05 - 0.51	0.05 - 0.43
85	93.45 - 94.71	0.00 - 0.18	9.57 - 11.63
95	87.42 - 89.51	7.23 - 9.60	12.73 - 15.68

1.4.2 JPEG Quality Variations

The testing that was produced for the quality variations produced numerous results. We begin by examining the baseline results that were obtained. Baseline testing was performed so that the expected accuracy could be seen for other quality levels. These are shown in Table 2. The baseline results show a high level of accuracy when training a model on a fixed quality regardless of the actual quality level. This high accuracy level allows us to see any reduction of accuracy easily when the quality of the test set is varied. Due to the variety of tests performed we discuss each test in the relevant section below.

Quality Variance on a 75 quality model

Table 3 shows the results obtained when training a model with 75 quality images embedded at 25% and then testing it on images with qualities of 55, 65, 85 and 95 all at 25% embedding. The results show that the 75 quality model was able to obtain almost equivalent accuracy for the 55 and 65 images. The accuracy of both of these was less than 1% away from the baseline results. The accuracy for the 85 quality images was lower than the respective baseline performance by 6% but the false negative rate increased by 10% which indicates that the increased quality

Table 4. Quality Variance against a 55 quality 25% embedding model

Quality	Overall Accuracy %	False Positive %	False Negative %
65	99.76 - 99.97	0.05 - 0.43	0.00 - 0.16
75	97.68 - 98.45	2.23 - 3.56	0.50 - 1.33
85	85.55 - 87.45	10.81 - 13.89	13.17 - 16.30
95	71.18 - 73.87	33.23 - 36.44	1.36 - 3.27

of the images led the model to class steganographic images as clean images. This is the opposite of the 55% model which experienced a slight increase in false positives which would indicate the poorer quality of the image led the model to conclude that a clean image contained hidden data. Finally, the 95% model experienced the highest loss of accuracy losing 12% overall with a high false and positive negative rate.

Quality Variance on a 55 quality model

The results from training on 55 quality images and testing against 65, 75, 85 and 95 quality images is shown in Table 4. These results show a decline in accuracy as the quality of the testing images increased. Although the accuracy for 65 and 75 are close to their baseline results the accuracy for 85 and 95 qualities is poor. In particular the testing of 95 quality images showed a large amount of false positives. This can be explained when referring back to the JPEG compression algorithm. The 55 quality uses higher values for the quantizer matrix which causes more loss during compression. This in turn leads to more visible blockiness in the image. The 95 quality images lose very little during the compression process due to having a quantization matrix of low values and thus do not have nearly as much visible artifacts as the 55 quality images. Because of this it means that the model is instead classifying the 95 images as containing hidden data due to the vast differences in the blockiness of the image.

Table 5. Quality Variance against a 95 quality 25% embedding model

Quality	Overall Accuracy %	False Positive %	False Negative %
55	70.61 - 73.27	34.05 - 37.43	0.00 - 0.00
65	75.18 - 77.82	30.16 - 33.41	0.00 - 0.00
75	80.87 - 83.47	24.42 - 28.06	0.00 - 0.00
85	96.71 - 97.71	4.24 - 6.03	0.00 - 0.33

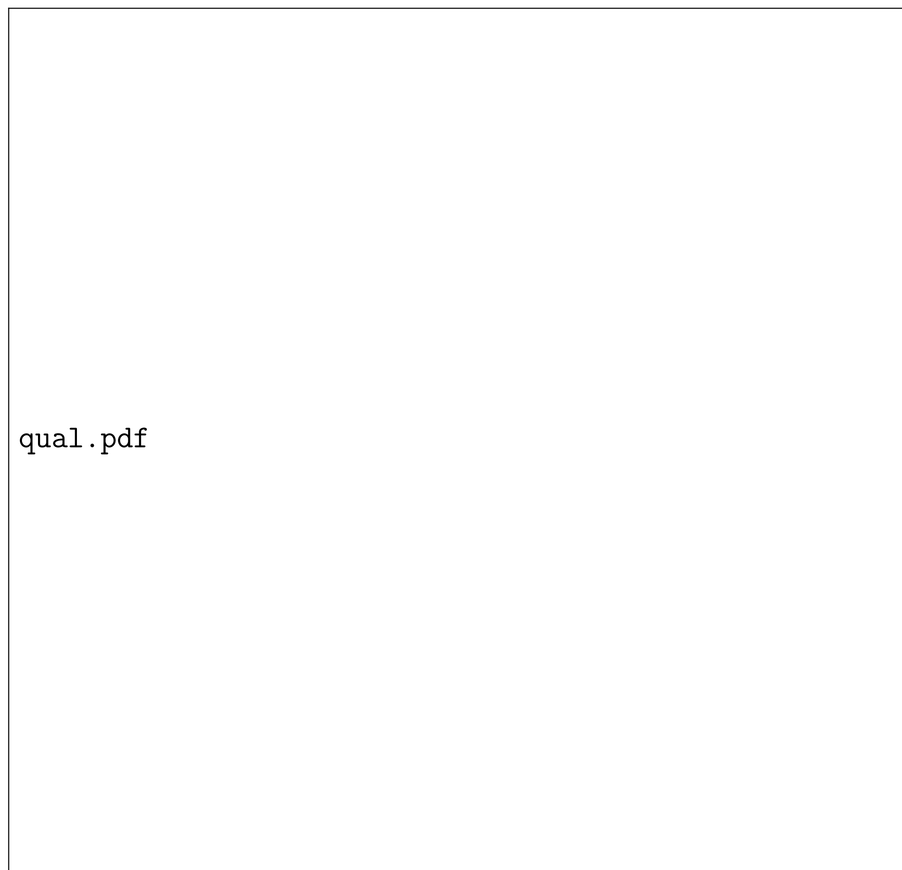

Figure 4. Quality variations on trained model versus baseline accuracy

Table 6. Testing of quality range 70 - 79 model

Quality	Overall Accuracy %	False Positive %	False Negative %
70 - 79	99.82 - 100	0.00 - 0.37	0.00 - 0.00
75	99.63 - 99.95	0.11 - 0.71	0.00 - 0.00

Quality Variance on a 95 quality model

The results in Table 5 show the opposite behavior of the 55 quality model in Table 4. The key difference however is that the accuracy immediately drops by a substantial amount on the 85 quality images. The accuracy loss as the quality gets further away from 95 can be explained in the same way as that for the 55 quality model. The images with 95 quality are of too high quality for the model to be able to classify other qualities in any reasonable manner.

Figure 4 shows the performance of the 55, 75, and 95 quality tests plotted against the baseline performance of the respective single quality models from Table 1.

Model trained on a 70-79 range of qualities

The final test was to see the accuracy when expanding the range of qualities that are trained on. The results presented in Table 6 show the results from two tests. The first presents the baseline performance of the 70-79 range of qualities. Clearly this result show that the range was highly accurate. Note that the bootstrap process here is instrumental in getting a good bounds on accuracy as the repeated sampling with replacement means that the testing set could contain a bias towards any quality in the range. The second test was to see how the accuracy of a single quality within that range changed when tested on the model. This allowed a direct comparison with the baseline performance of the 75 quality images in Table 1. The results show that the overall accuracy was less than 1% different and more importantly there were no false negatives. This indicates that

Table 7. Matrix of Embedding Variants against single embedding model

	5% Model	15% Model	25% Model	50% Model
5%	-	13.17	0.84	0.21
15%	99.74	-	78.07	0.47
25%	99.95	99.90	-	10.66
50%	99.84	99.95	99.84	-

the quality range model can detect a single quality within its range with the same accuracy as a model trained specifically for that quality.

1.4.3 Embedding Percentage Variations

Table 7 shows the matrix of results obtained from the embedding percentage variant tests. The table can be read by seeing the embedding percentage that was used to build the model along the top with results obtained from testing on the various other embedding percentage image sets vertically. We did not provide results for testing against a model built with the same embedding percentage as this would be a duplicate of the results given in Table 1. The results show the accuracy percentage of detecting images that have steganographic data at that embedding percentage. Because we are testing only with steganographic images and not clean images in this section we do not provide false positive values and note to the reader that false negative values are easily obtained by subtracting the accuracy from 100%.

From the results a clear pattern is apparent. For each model we create we are able to detect steganographic content with an embedding percentage that is higher than what was trained on to a high accuracy but unable to detect any content that is embedded at a percentage below the training set. In many cases image sets with higher embedding percentages than those trained on are detected with the same accuracy as the models for those higher embedding percentages are. We can see the contrast with embedding variations by comparing the 5% and 50% training

models. For the 5% training model we obtain very good accuracy on all of the subsequent test sets which are all higher than 5%. However, for the 50% model we have very low accuracy with almost no steganographic content detected for the images embedded at 5% when tested on it.

Also of note is the more extreme effect on low embedded images being tested when they are below the embedding percentage of the model. When the 5% images were tested against the 15% model we obtained a very low accuracy of just over 10%. Compare this however to the result from 15% images being tested against the 25% model. In this case we obtain an accuracy in the high 70% range which while not suitable for actual detection is much higher than the accuracy of the 5% images. This is important to note because in both cases the embedding percentage was 10% lower than the training set but much different results where obtained. Although some of this loss of accuracy is due to the fact that 5% embedded images are harder to detect than 15% embedded images, as seen in Table 1, there is a greater than 50% difference between the accuracies obtained in the embedding variation test but the baseline results show just an 8% difference when trained and tested on single embedding percentages.

1.5 Evaluation

The results obtained from section 1.4 give us the insight to make specific recommendations and "best practice" guidelines to allow software companies, law enforcement or academia to construct models that are suitable for real-world steganography detection. These guidelines are based around the smallest detectable level that is suitable for use in a real-world application. Various different uses for such a steganography detection tool may mean that some agencies require different levels of accuracy so therefore we base our recommendations so that different accuracy needs can be met. We describe the guidelines for each of the

aspects examined in this paper in the following sections.

1.5.1 JPEG Quality Variation

The work shown in section 1.4.2 has several insights into model construction and thus allows us to build guidelines around our results. Figure 4 clearly shows that it is not suitable to merely train on a single quality and then expect accurate results from the model for other qualities. This is particularly relevant when training on either high or low quality levels, 95 and 55 from our tests, as these have been shown to produce poor results when testing other quality levels. However, we also observe that in table 3 that the single 75 quality model performed well for all quality levels that we tested against it. The reason for this improved accuracy compared to the 55 and 95 quality tests is because the quality of the 75 images sits between the poor quality of 55 images and the high quality of 95 images. Because of this the 75 quality model does not see such an extreme difference in image quality as the 55 and 95 models do. Even though the results were good for the 75 quality model we do not recommend its use in practice. The reason for this is that although the results are strong in our testing in practice images could have qualities that are as high as 100 and as low as 1. Because of this the 75 quality model would suffer from the same problem due to the dramatic quality difference in those images.

From the results of the quality range model test, shown in 6, we recommend that models built for use in practice follow this approach. The results of a model trained on qualities from 70-79 showed excellent results in both its baseline testing and being tested against just 75 quality images. This means that we achieve two key goals by building models in this way:

High Accuracy - the results show that we are able to still achieve high accuracy when training on a range of qualities even when allowing a bias to

be introduced. This was shown by both the baseline and single quality tests performed.

Reduction of Models - because the results indicate that we can use a model with trained on a range of qualities we can reduce the number of models that have to be trained which has the effect of reducing engineering time of a product. This is highly beneficial due to the time and space requirements of creating thousands of images that are of a given quality and contain steganographic content at various embedding levels.

These outcomes have allowed us to suggest the following guidelines with regards to model construction with JPEG quality in mind:

1. Construct a data set of clean and steganographic images that have been compressed to a random quality level between two bounds.

2. Partition this set into training and bootstrap sets.

3. Construct the model and perform bootstrapping to determine baseline accuracies.

4. If the accuracy is acceptable then the range of qualities could be expanded and this procedure repeated. Alternatively construct additional models using the same range but at quality levels so that models are constructed to cover the JPEG quality range. For example, if a range of 70-79 is acceptable then construct models with ranges 50-59, 60-69, 80-89, 90-100. We believe that a model trained on images with quality levels of 50 will adequately detect all qualities lower than it due to the quality at 50 being very poor.

5. If any of the subsequent models to not perform well then the range should be shrunk for that particular range. For example if a range of 10 produced the

required accuracy at all levels apart from 90-100 then we would reduce the size of that particular range. If acceptable quality was found for it within a range of 5 then we would produce models for 90-94 and 95-100.

From our testing of both ranges of qualities and of a single quality we believe that a range of 10 quality levels provides the best compromise between accuracy and model reduction. The reason for this recommendation is that the results indicate a drop of accuracy once the quality level is greater than 10 away from the trained model.

1.5.2 Embedding Percentage Variation

From the results in section 1.4.3 have shown that we have two distinct and interesting properties found in the models we construct. The first is that we cannot accurately detect any steganographic content that is below the embedding percentage we have trained on. This is clearly visible from the results given in Table 7 which show for any of the models we construct that when detecting steganographic content leads to classification, namely a false negative, of the image. Therefore in real world use it is critical that the smallest possible embedding level that is deemed to be detectable has a model created specifically for it. This can be easily achieved in by following some basic steps:

1. Construct a data set and partition it into clean and steg training sets along with clean and steg bootstrap sets.
2. Using the steg training and bootstrap sets embed these at a given percentage.
3. Train the model and then test by using the bootstrap method.
4. If the accuracy is acceptable then reduce the embedding percentage and repeat until the bootstrap results give an accuracy that is at the accuracy threshold.

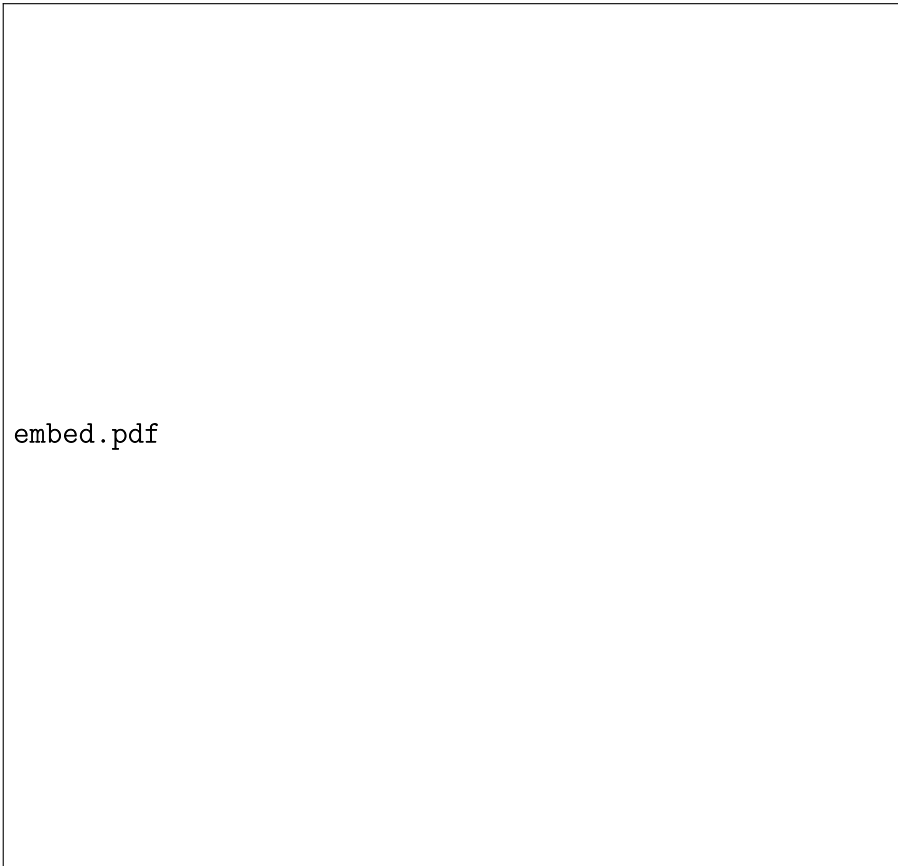

Figure 5. Graph of accuracy for 5% embedding model when on other embeddings (solid line) versus baseline performance (dashed line) for single embedding models

Completion of this process will give the smallest possible embedding amount that can be accurately detected using the steganography detection tool. The process does this because we have shown that we cannot detect steganography well below a model and the bootstrapping method gives you the accuracy range of the model with bias so the process allows a tool to be developed with its restrictions on detecting well known.

The other key result from the work in section 1.4.3 is that we could see that we obtained good accuracy when testing steganographic images with higher embedding percentage than the trained model. This result is highly useful when producing a tool for real-world usage as it allows us to potentially reduce the num-

ber of models that an image would have to be tested against. If we construct a product that contains m models for embedding percentage then for each image we would have to run it through the testing procedure m times. This would be the case for any clean image, unless a false positive classification occurs, as every model would classify it as clean and so it would be tested by all m models. This causes a considerably performance hit when testing a large number of images. Therefore we want to keep the number of models that an image would be tested on as small as possible. The results from section 1.4.3 allow us to reduce the number of models we need by using the fact that a model trained at one embedding percentage can detect images higher than it. Because of this we have established the following guidelines:

1. From the work above there exists a model trained on a given percentage that meets the accuracy specifications required.

2. Bootstrap this model on bootstrap sets composed of images at higher embedding percentages than the model.

3. Ensure that the accuracy for the higher embedded images still meets the tools requirements.

This process will allow a reduction in the number of models required. This will reduce the amount of time required to test images for the law enforcement agency. The reason we can reduce the models and still maintain the desired accuracy is because of the bootstrapping method. An example of this is given in Figure 5. The dashed line shows the minimum accuracy of the model, taken from table 1, and the solid line the results from table 7 showing the testing performed on the 5% model. It is clear that the accuracies shown are correlated with little deviation from each other. From the figure we know that we can detect, for example, 50%

embedded images just as well as the 50% trained model. The important part from an application engineering perspective is that if we reduce our models down to, as in the example, just the 5% model then we also accept the false positive accuracy from the 5% model. As we have calculated this in the guidelines we are already aware of it and a decision can be made if this is an acceptable false positive accuracy for the applications intended use.

1.6 Conclusion

In this paper we have presented two key variations that would be found in real-world use of a tool to detect steganography. From our testing we have shown some flaws in these assumptions and some interesting results that are relevant in the model construction phase of engineering such a tool. These results have allowed us to make some recommendations and guidelines for construction of models used in steganography classification.

We have shown that JPEG quality is an important factor when detecting steganography. Due to the wide range of qualities that exist in JPEG images we cannot simply assume that all images will be of a fixed quality. We have shown that accuracy suffers as a result of making this assumption. Furthermore we have also shown that certain choices of image quality make for worse classification in this instance. Both 55 and 95 quality models exhibit very poor accuracy when testing on images that have different qualities. In order to avoid having to create a model for every quality level we instead tested the effect of training on a range of qualities. The results from this gave high levels of accuracy which allowed us to create fewer models that still had a high level of accuracy. This is critical in terms of engineering a tool as it vastly reduces the time and space to train such models.

The second variation we examined was pertinent to variations in embedding percentage of steganographic images. These results showed that models were un-

able to accurately detect steganography content less than the amount they were trained on, but inversely were able to detect content above it. This again has the key ability of reducing the number of models that need to be produced, but also has the benefit of meaning the tool does not have to be run on as many models during use. This is crucial in reducing the time that the tool takes to run in the hands of law enforcement agencies.

List of References

[1] G. Kessler, "Steganography: Hiding data within data," September 2001. [Online]. Available: http://www.garykessler.net/library/steganography.html

[2] T. H. Barr, *Invitation to Cryptology.* Prentice Hall, 2001.

[3] R. J. Menon, "MP3 Steganography and Steganalysis," Master's thesis, University of Rhode Island, 2009.

[4] J. P. E. G. (JPEG), *JPEG ISO/IEC 10918-1 ITU-T Recommendation T.81.*

[5] G. K. Wallace, "The jpeg still picture compression standard," *Commun. ACM*, vol. 34, no. 4, pp. 30–44, 1991.

[6] J. J. Fridrich, "Feature-based steganalysis for jpeg images and its implications for future design of steganographic schemes," in *Information Hiding*, 2004, pp. 67–81.

[7] A. Westfeld and A. Pfitzmann, "High capacity despite better steganalysis (F5–a steganographic algorithm)," in *Information Hiding, 4th International Workshop*, vol. 2137, 2001, pp. 289–302.

[8] N. Provos and P. Honeyman, "Hide and seek: An introduction to steganography," *IEEE security & privacy*, vol. 1, no. 3, pp. 32–44, 2003.

[9] S. Hetzl, "Steghide."

[10] P. Sallee, "Model-based steganography," *Lecture Notes in Computer Science*, pp. 154–167, 2003.

[11] H. Farid, "Detecting steganographic messages in digital images," Dartmouth College, Hanover, NH, USA, Tech. Rep., 2001.

[12] Y. Q. Shi, C. Chen, and W. Chen, "A markov process based approach to effective attacking jpeg steganography." in *Information Hiding*, ser. Lecture Notes in Computer Science, J. Camenisch, C. S. Collberg, N. F. Johnson, and P. Sallee, Eds., vol. 4437. Springer, 2006, pp. 249–264.

[13] T. Pevný and J. Fridrich, "Merging Markov and DCT features for multi-class JPEG steganalysis," in *Proceedings SPIE, Electronic Imaging, Security, Steganography, and Watermarking of Multimedia Contents IX, San Jose, CA, January 29–February 1*, E. Delp and P. Wong, Eds., vol. 6505, January 2007, pp. 03–14.

MANUSCRIPT 2

TCP Steganography

2.1 Abstract

Steganography exists in network traffic by using the medium in a way it was not designed for. We form covert channels of communication and create steganographic content by making the network communication appear as regular network traffic. We have developed a novel approach which is based around purposely corrupting TCP packets. We do this by selecting a packet which is intended for the host we wish to communicate with and then embedding the hidden data into it, but modify the packet so that it has an invalid checksum. This means that the network stack on the other host will silently drop the packet, but a specially designed module will intercept the packet before it is dropped and will retrieve the data. Because we corrupt a TCP packet the packet will be sent across a routed network as devices at this level will not check the TCP checksum. We evaluate the technique and present its strengths and weaknesses. This work is important as it alerts law enforcement agencies to the potential use of corrupt packets to communicate covertly.

2.2 Introduction

Although digital steganography is typically discussed with regards to visual or audio methods it is also possible to use steganography in computer network communications. Within networking this type of steganography is commonly referred to as a *Covert Channel*. The main difference between classic steganography and covert channels is that a covert channel hides data by using a system in a way it was not originally designed for[1]. Computer network steganography, or network covert channels, is an established field that presents many problems for law enforcement

agencies. In addition to the threat of criminal communication that steganography presents, network based steganography also makes investigation much more complex. The reason for this is that should communication be taking place across a network then the evidence of it may only exist for the lifespan of the network packet. If you compare the lifespan of a transferred JPEG containing steganography then it is easy to see that the JPEG may likely exist on at least two machines - the sender and receiver. However, in network based steganography the network packet is not typically stored on any machine. The data may be extracted and reside on one machine, but the evidence of its communication method does not exist unless the packet is actively captured by some law enforcement agency. Because of this it is vital that law enforcement agencies be aware of any possible methods of covertly communicating across a computer network.

In this paper we will look at a new technique for network steganography that allows communication to take place over a routed network using a technique based on the Transmission Control Protocol (TCP), when used in the TCP/IP suite, that makes the packet appear that it contains a network error. This is achieved by purposely corrupting the packet by modifying the checksum field in the TCP packet. The biggest strength of this techinque is that it has the ability to communicate over a large routed network, such as the Internet.

We begin our discussion by examining how communication takes place when using TCP along with the IP network layer protocol. Following this we discuss the method we use to hide data. Finally we evaluate this steganography method and look at its advantages and disadvantages.

2.3 TCP/IP

TCP is specified in [2] as a reliable process-to-process communication service which when combined with IP provides the common TCP/IP communication

method that is used on the Internet. TCP is a connection orientated protocol[3] but only in terms of the end systems involved. TCP typically runs over packet-switched networks which use IP. TCP establishes a full duplex connection between two hosts over a tuple of port numbers. We begin by describing the contents within the TCP packet. Figure 6 shows the contents of the TCP packet.

2.3.1 TCP Header

Source & Destination Port - Contains the source and destination port numbers to identify process-to-process communication.

Sequence Number - When a connection is initiated an initial sequence number (ISN) is selected. Further on whenever data is sent the sequence number will be set to the first byte in the current stream in addition to the ISN. This sequence number is used to acknowledge that the data was received.

Acknowledgment Number - TCP packets acknowledge data received in order to achieve reliable transfer. TCP does this by sending acknowledgments, or ACKs, with the acknowledgment number set to the next byte that it is expecting. Upon receiving the ACK the sender knows what has been received successfully.

Data Offset - The Data Offset field specifies the length of the TCP header so that the client knows where the data starts in the packet. This is required because TCP Options can be of variable length.

Reserved - 6 bits are reserved and must be zeroes.

Flags - The flags consists of a bit field of 6 bits that specify (in order) Urgent Pointer, Acknowledgment, Push function, Reset, SYN or Synchronize, and

FIN or termination of connection.

Window - The window field specifies how much data the receiver is expecting to handle. This allows the sender to send that much data before expecting any acknowledgments of the data being received. This allows a TCP connection to be much faster as it can send multiple packets without waiting for individual packet acknowledgments. The window field is adjusted by the receiver and allows flow control for the TCP connection.

Checksum - Holds a checksum for the entire header and data. The checksum is typically calculated by forming a pseudo-header that contains all TCP header information, data section, and is prepended with the source and destination IP address from the IP header. A ones complement is taken over each 16 bit word in the pseudo-header and is padded with zeros if needed.

Urgent Pointer - If the Urgent Pointer flag is set then this points to the sequence number of the data that follows the urgent data.

Options - A variable length field that can hold different options for the TCP connection. Is typically used during initial connection setup to set Maximum Segment Sizes and during the TCP data transfer to hold timestamps that allow for congestion control.

Padding - The TCP header must end on a 32 bit boundary so padding bits may be used.

Data - Contains the data for the packet.

2.3.2 TCP Connection Establishment

As already noted TCP is a connection-orientated protocol. In order to establish a TCP connection between two hosts they must go through a three-way

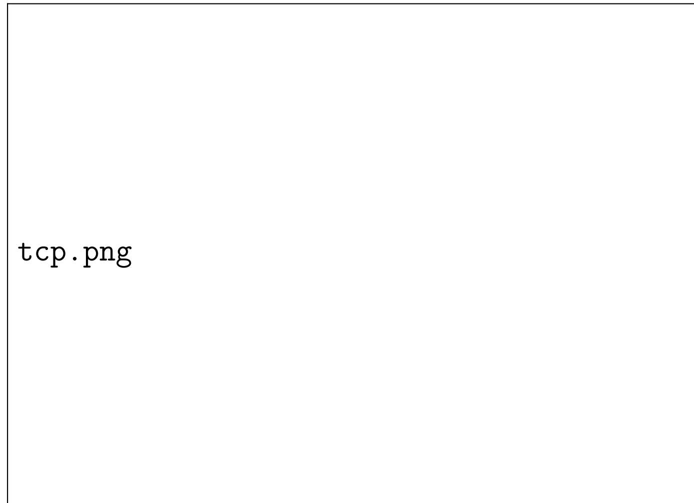

Figure 6. TCP packet specification

handshake procedure[3].

1. **SYN Packet** To begin the client host sending a synchronize, or SYN, packet to the server. The packet at this point contains the destination port of the server, the source port which is decided by the client's operating system, a sequence number that is randomly selected, and the SYN flag set to 1 in the flags part of the packet.

2. **SYNACK Packet** When the server receives the SYN packet from the client it then replies to the client with a SYNACK packet. The server will set its destination port to that of the source port and vice versa based on the relevant fields in the SYN packet received. The server will also set both the SYN and ACK fields to 1 in the flags section of the TCP packet. The server will set the acknowledgment number to the sequence number received in the SYN packet incremented by 1. In the same way the client randomly selected an initial sequence number the server will do the same in order to set its sequence number. This packet is then sent to the client.

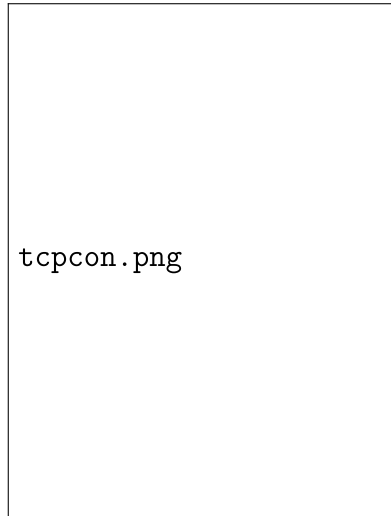

Figure 7. TCP Connection Establishment

3. **ACK Packet** The final step in order to establish the connection is for the client to acknowledge that it received the SYNACK from the server. This is achieved by sending a packet with the ACK flag set to 1 and the acknowledgment number set to the servers sequence number incremented by 1. The sequence number is also incremented by 1 in the packet. Once this packet is sent and successfully received by the server then the connection is considered established.

An example of the TCP connection establishment procedure is shown in Figure 7.

2.3.3 Reliable Delivery of TCP packets

TCP is a reliable data transmission data protocol. This means that it ensures that all data is delivered to the end host. TCP achieves this by the use of timeout windows and acknowledgment packets[3]. As mentioned above, when a host sends a TCP packet it gives the packet a sequence number. This sequence number is generated initially by the operating system but then incremented by the amount of

data sent in each packet. The receiving host uses the sequence number in the packet it receives to indicate what the last byte of successfully received data was. Due to the nature that a hosts acknowledgment packet could be lost TCP maintains a timeout window. If a sender does not receive an acknowledgment packet within the timeout window then it will resend the original packet for which it did not receive an acknowledgment for. An example of this approach is given as follows:

- Host A sends 8 bytes of data to Host B. Host A has a sequence number of 52.
- Host B receives the data and responds with an acknowledgment packet with an acknowledgment number set to 60. This is because it is indicating it received all 8 bytes correctly.
- Host A sends 4 bytes of data to Host B. The sequence number this time being 60.
- This packet is lost during communication. The timer window for this packet expires.
- Host A resends this packet.
- Host B receives the packet and acknowledges it with its number set to 64.

This example is also shown in Figure 8. Although this example only covers a simple example it illustrates that TCP ensures delivery of packets. Further information on different packet delivery scenarios can be found in [3]. The reliability of TCP data delivery is critical to the success of our steganographic technique as it ensures that the intended network communication is successful even after we purposely corrupt a packet.

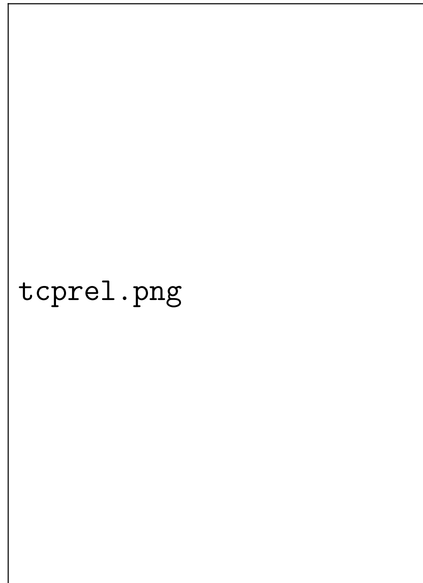

Figure 8. TCP Reliable Delivery with lost packet

2.4 TCP/IP Steganography

Data hiding has long existed in the entire OSI network stack[4, 5] and is therefore expected in TCP. Various methods have been put forth to send data covertly using TCP. Some of the basic approaches involve simple packet manipulation which include techniques such as setting flags, like the Push flag, in order to communicate a series of binary data[1]. One of the first sophisticated approaches to TCP steganography was suggested by Rowley in [6]. The approach used by Rowley is based upon purposely selecting the initial sequence number (ISN), as discussed above, of the TCP connection. By purposely selecting an ISN you are able to hide some data in the least significant bit of the ISN. Clearly this method requires creating many connections in order to send all the data. For example, to send 1KB of data you would need to create 8192 connections as only 1 bit can be sent per connection. Because many implementations of the TCP/IP stack use a carefully constructed ISN in order to prevent overlap and improve security in subsequent connections this basic substitution method can be detected by use of

a SVM[7]. The fact that this method also requires initiating a lot of connections means that the SVM method can readily detect it in use. In similar fashion Giffin et al developed an approach using TCP Timestamps[8]. They do this by purposely delaying processing on the TCP packet by some amount. Because this technique introduces a timestamp which is essentially inaccurate and of no use for calculating TCP windowing it creates an effect by which the lower bit of the timestamp is essentially random and thus can be modified in order to hide data.

2.5 Hiding data in corrupt TCP packets

In this part of the project we develop a new data hiding technique based on purposely corrupting the TCP packet. This approach is based upon the work by Szczypiorski in the HICCUPS project[9]. Szczypiorski develops a technique based on hiding data in the data link layer of the OSI model. This is achieved by purposely setting the checksum to be incorrect of the data link layer header but still containing a secret key known to both the sender and receiver. Due to the broadcast nature of the data link layer all machines on the broadcast domain will receive this packet, but due to the corrupt checksum will merely drop the packet. The machine that is expecting to receive hidden data will examine the checksum to see if it contains the secret key and if so will process the packet and obtain the hidden data. The HICCUPS project uses a wireless network due to the increased number of corrupt packets present compared to a wired network.

One of the drawbacks of the HICCUPS approach is that it is strictly limited to a local area network (LAN). This means that HICCUPS cannot reach any hosts that are outside of its broadcast domain. This severely restricts the physical distance that two parties can be apart to the order of 100's of meters. Due to this limitation we have developed a technique involving corruption at the TCP transport layer. The immediate benefit of doing this is that any two hosts can

Figure 9. TCP connection between two hosts showing how a corrupt packet can be injected into the normal stream of communication

communicate across a routed network, and therefore traverse broadcast domains, by hiding data in what appears to be a corrupt TCP packet. The reason that this technique is able to be used across a routed network is because it operates at the transport layer and routing devices work at the network layer, on IP in a TCP/IP network, so the TCP checksum is not checked until it reaches the end host. Figure 9 shows the general approach of inserting corrupt packets into a TCP session. Furthermore this technique is does not introduce a perceptible disruption to any user. As described in section 2.3.3 TCP is a reliable transmission mechanism and it uses acknowledgments to tell the sender what has been successfully received. When a host receives a TCP packet with a corrupt checksum it will send back an acknowledgment packet indicating that it did not receive it. Therefore the host will resend the packet once again. Introducing a corruption merely causes a TCP packet retransmission by the sender. We do not lose any information and thus do not introduce any altered behavior noticeable via packet sniffing or network application.

The corrupt TCP steganography technique allows for a variety of data embedding possibilities. We introduce two possible methods in this paper - a low and high capacity technique. In the low capacity technique we can use the corrupt checksum to send a byte of hidden data in each corrupt packet. We require the use

of a key so that we can determine that data is sent from the intended party and not just an actual TCP corruption. The key takes up a byte and the checksum is 2 bytes long. So therefore in each corrupt packet we can embed a byte of data. The obvious drawback to this low capacity technique is that it would require sending numerous corrupt packets in order to communicate even a small message. The high capacity technique uses the data segment of the packet to place data. This allows us to hide in up to 1.4KB[3] of data per packet. This means that we may only need to send a single corrupt packet in order to communicate. The disadvantage of the high capacity technique is that careful inspection of the corrupt packets will clearly show that the data is noticeably different from the data that would be sent in the retransmitted packet. In the low capacity technique the corrupt checksum itself contains the data so knowledge of this technique would be required to decipher the data. Therefore although the high capacity approach requires less corrupt packets to be sent should packet sniffing take place then it would be easily spotted. In this paper we have implemented the low capacity technique and evaluate the steganography tool with this in mind.

The technique requires that there be two hosts actively participating in the communication. One host will be the sender and the other the receiver. We describe the technique of both hosts.

2.5.1 Sender

The sender application is described below, based on the low capacity approach, and is shown in pseudocode in Algorithm 1.

Initialization The sender program is loaded and initialized with the destination IP address, destination TCP port, secret key and the data to be hidden.

Packet Selection The sender program is able to scan all outgoing packets on the system. A scan of the packet will tell it if the packet is a TCP packet and whether its destination port and IP address matches that of the initialization values. If it does then this packet becomes a candidate for corrupting. On a candidate packet we then check that it is not part of the connection establishment, as mentioned in section 2.3.2, as this would stop the connection from being established which does introduce a noticeable delay to a user of that network application. If the candidate packet is part of the regular TCP data stream then we decide randomly whether or not to corrupt it. Selecting the packet in this random manner again makes the steganography appear more random and much more like a realistic corruption event.

Packet Corruption Once a packet is selected we can begin the process of corrupting the packet and inserting the hidden data. To do this we take the 8 bit key and store it in the upper 8 bits of the checksum field. The remaining 8 bits are filled with a byte of steganographic data. The upper 8 bits allow the receiver to know that this corrupt packet is intended for them. A final check is made to ensure that the checksum we created actually is incorrect for the packet.

Send Packet At this point the corrupt packet is formed and contains hidden data. We then allow this packet to be sent using the operating systems standard send functionality.

TCP Timeout Because the receiving host will receive a corrupt packet the behavior of TCP is that the receiving host will either not acknowledge that packet or send a duplicate acknowledgment packet. This means that

the sender should resend the packet once the timeout window is reached or the duplicate acknowledgment is received. At this point we allow the operating system to send the the original packet as this will continue the regular network communication between the hosts.

Repeat We continue this process until all steganographic data has been sent.

The pseudocode for the sending algorithm is as follows:

Algorithm 1 Sending corrupt TCP packets

```
1: procedure TCPCorruptSend(DestIP,DestPort, Key, HiddenData)
2:     dataSent ← false
3:     while dataSent ≠ true do
4:         packet ← getOutgoingPacket()
5:         if packet == TCP then
6:             pDestIP ← packet.destIP
7:             pDestPort ← packet.destPort
8:             if pDestIP == DestIP && pDestPort = DestPort then
9:                 pType ← packet.type
10:                if pType ≠ SYN && usePacket() == true then
11:                    packet.checksum ← Key&data
12:                    sendPacket()
13:                    dataSent ← isAllDataSent()
14:                end if
15:            end if
16:        end if
17:    end while
18: end procedure
```

Notice that the ability of this program to wait until it finds a suitable packet allows it to work in a passive manner. It does not actively generate traffic but instead waits for some user or process to generate traffic that can be modified to contain the steganographic data. This allows traffic usage to appear very "natural" as all traffic is generated by user applications.

2.5.2 Receiver

The application to receive packets on the receiving host is very simple. The key to receiving the corrupted packet is that we must be able to read the packet before the operating system discards it. This is easily done in the same way that many packet sniffers operate, such as Ethereal or tcpdump[10], by loading themselves into the network stack of the operating system such that every packet passes through them. We can create such a program which works by looking for corrupt TCP packets. We describe the algorithm and provide its pseudocode below:

Initialization We load the receiver program with information on the IP address that the sender is using, the TCP port that the sender will be sending corrupt packets to, and the secret key that is used to identify the corrupt checksum.

Packet Scanning In order to receive the corrupt TCP packet we need to load the program into the network stack so we receive all packets before the operating system discards them. From this we scan each packet that arrives to see if it is a TCP packet. If it is then we check the senders IP address and the destination port in the packet to determine if this packet is a possible steg packet. If this information matches the initialization data then we extract the checksum from the packet and see if the packet is in fact corrupt.

Key Check Given that we have a corrupt packet that is on the channel that we expect corrupt packets we need to ensure that it contains the secret key that is agreed upon by both users. If we did not perform this check we may receive a packet that has been corrupted but does not actually contain any steganographic data. To check this we extract the upper 8 bits from the

checksum. We then compare this to the key given from the initialization step and if they match then we have received a packet with steganographic data.

Data Extraction To extract the data we simply take the lower byte from the checksum. This data can then be outputted to the user on the host.

Algorithm 2 Receiving corrupt TCP packets

```
1: procedure TCPCorruptRecv(SenderIP, SourcePort, Key)
2:     Load into network stack
3:     repeat
4:         packet ← getIncomingPacket()
5:         if packet == TCP then
6:             pIP ← packet.destIP
7:             pPort ← packet.destPort
8:             if pIP == SenderIP && pPort == SourcePort then
9:                 checksum ← packet.checksum
10:                if isCorrupted(checksum, packet) == true then
11:                    packetKey ← get8bits(checksum)
12:                    if packetKey == Key then                ▷ Is a steg packet
13:                        data ← packet.Data
14:                        output data
15:                    end if
16:                end if
17:            end if
18:        end if
19:    until User terminates
20: end procedure
```

2.6 Implementation

In order to develop a proof of concept we use the approach that Rutkowska et al use in their NUSHU steganography tool. The NUSHU tool is based around Linux kernel modules and *protocol type handlers*[11]. The Linux operating system allows modules to be loaded in to the kernel. Although this is typically done for device drivers we can use kernel modules to perform any function we desire. In addition, the Linux network stack operates by registering a series of protocol

type handlers that it calls upon to handle network traffic. Combining a kernel module with a protocol type handler we are able to create our corrupt TCP packet steganography tool.

The protocol handlers in Linux consist of a function that gets a clone of a packet inside of a *skbuff*, or socket buffer, structure. Linux uses these skbuff structures to place each packet in and the structure consists of pointers to the various network headers, the data, and variables that instruct the protocol handler on its responsibilities with regards to checksumming and so forth[12]. It is important to point out that any protocol handler is given a clone of the packet and as such is limited as to what changes can be made to the packet. A protocol handler is registered with the kernel by creating a *packet_type* structure which indicates to the kernel what function should be called and to what packets the protocol handler is interested in seeing. Once this is created the protocol handler is registered into the kernel using the call *dev_add_pack* which assigns the function to a list[12]. When the network stack receives a packet it looks on its list of protocol handlers and finds which functions it needs to call. At this point it calls each function with a clone of the skbuff structure.

We harness this ability in our kernel modules for both sending and receiving of hidden data. For the sending program we create a protocol handler that examines each outgoing packet. If the packet matches the conditions required to corrupt, see Algorithm 1 for details, then we randomly select the packet. If the packet is chosen we modify the checksum to include the key and a byte of hidden data. As our modifications are complete we simply return from our function as the operating system will send the packet automatically for us.

The receiving module operates in similar fashion, but instead is registered to receive all incoming IP packets. From this if the packet matches the required

conditions then its checksum is computed and compared to the checksum that was received. If the packet is corrupt then we extract the upper byte and compare it to our key. If the key matches then we extract the lower byte and output it as received data. We then return from the function and allow the operating system to either drop or send the duplicate acknowledgment packet on our behalf.

One additional feature common to both programs is the use of the "proc" filesystem. This file system exists to allow program interaction from within a file like structure. We create entries in the proc filesystem for both tools that allow us to pass data in so that it may be sent and to view any received steganographic content.

The source code for both these programs is given in Appendix A.

2.7 Evaluation

We will evaluate the tool under four key areas for steganography tools. The areas of evaluation are all critical to a successful steganography tool. These areas are presented in the following sections.

2.7.1 Imperceptibility

In order to measure the imperceptibility, that is how much its usage is noticeable, we need to evaluate from the users aspect. If a user is using some network service that we happen to be hiding steganography in, such as HTTP or SSH, then we must look to see if corrupting TCP packets used in these communications is noticeable to the end user. Our design approach has made it so the user notices very little even when corrupt packets are being sent. No error messages are communicated to the user, and furthermore any expected behavior from the network application is produced. This is achieved because we never modify or lose any of the data that is sent. The TCP retransmissions ensure that all data is delivered.

The only noticeable effect to the end user is in an additional delay due to the retransmission of the packet. However, as long as corrupt packets are not being sent at a high level then the retransmission time, on the order of microseconds, is negligible to the user. Therefore our tool has a high level of imperceptibility which is vital in a steganography tool.

2.7.2 Undetectability

Due to the high level of imperceptibility our tool provides detection of it must be done at the network or system level. Assuming that the steganography implementation we have created is used on machines where the sender is the systems administrator then the only way of detecting the usage of this tool is by some third party sniffing the traffic between the hosts. However, this process is made difficult by several design principles in the tool

- **Capturing Network Evidence** - this steganography tool differs vastly in that the evidence of its use only exists while the packets are on the network and, due to the usage of the proc filesystem in our implementation, when the machine is active. The Linux proc filesystem is only active while the machine is running so any forensic acquisition with the intent on capturing evidence of the tools use has to do it while the machine is running. This is much harder than the usual forensic approach of imaging the physical hard drive. The same approach has to be taken when capturing evidence of the tools use across the network. If the packets aren't captured when they are sent then the evidence is lost permanently. Both of these issues make the job of detecting its use by law enforcement considerably harder.

- **Selection of port** - because the tool is able to be used over any TCP port it makes detecting it much harder as it requires either knowledge of the tools

exact use, provided by say an informer, or that all network traffic must be sniffed. This causes additional difficulty for law enforcement as sniffing all traffic between two hosts could contain a lot of traffic that does not have steganographic content in. For example if the steganography tool was used on SSH but the machine also had frequent HTTP requests then the packet sniffing will have both protocols in. Not only does this increase the amount of time spent analyzing the packet sniff but it also has the issue of space requirements due to the large number of packets being detected.

- **Time Window** - due to the passive way in which the tool operates, namely that it waits until it see certain traffic in order to send hidden data, it allows the operators of the tool the ability to pick random times in which to send the steganography. This means that a criminal could use it over a period of time, or could wait a number of days. Again this makes law enforcements work harder as there may be no guarantee on when the data is sent. The consequence of this is that any packet sniffing would have to be done over a longer period of time which again requires increased analysis and space.

- **Packet Analysis** - the corrupt packets that are sent essentially make the network communications appear regular. The only behavior that is noticeable is that corrupt packets were received. It would be up to the investigator to examine the number of corrupt packets and to determine if these are possible steganography packets or whether they were in fact genuine corrupt TCP packets.

Although the above points show that the tool is not trivial to detect it currently does have some weaknesses that may allow it to be detected. One of the key weaknesses in the implementation given is that all corrupt packets contain the same key. The key is visible, in the upper byte, of the checksum and as such if

a series of corrupt packets are sent and captured then an investigator may notice that the packets all contain the same value in the upper byte. This may indicate the usage of steganography and from there decoding of the steganographic data.

2.7.3 High Capacity

For the implementation given in this paper we developed the low embedding method. Because of this we only send one byte of hidden data per packet we corrupt. Obviously to send a large amount of data then we must corrupt possible thousands of packets. This means that there is more chance of the tools usage being detected. However, in this paper we have also suggested a technique for embedding a high amount of data wherein we use the data section of the TCP packet. This means that we would only have to send one packet for data sizes of up to 1.4KB. Although we send less corrupt packets in this approach we do drastically change the data contents of the packet. This in itself may raise awareness of the tool via packet sniffing. However, if we are embedding binary data and we use a protocol that uses encryption then the changed data section may not be noticeable.

2.7.4 Robustness

Clearly it is important that any steganography tool be robust so that hidden data is not lost during transmission. Due to our tool being based around network transmission then we have the following situations that can cause the robustness of the tool to be compromised.

- **Changes to the Checksum** - because our technique relies on purposely changing the checksum of the TCP packet we clearly need to maintain this checksum throughout delivery of the packet. On a routed network the TCP header is never examined due to the router having no need of checking it so in general the checksum is never modified during routing. The one exception to

this is if Network Address Translation (NAT) is taking place. NAT converts one IP address to another and is typically used so that a single IP address can be used on the Internet rather than assigning each host on a LAN an IP address. In order for NAT to take place the NAT device modifies the packet. It changes the senders IP address, as well as the source port within the TCP header. These changes mean that the TCP checksum be recalculated. Most devices will perform a partial computation of the checksum where the checksum in the packet is used in conjunction with the new information in order to calculate the checksum. This presents a problem for our tool because the checksum we purposely inserted has now been changed. Therefore at best the hidden data is lost, but at worst the hidden data could have been modified. Therefore the tool is not suitable for use where NAT is in place.

- **Lost packet** - although no router will drop the corrupt TCP packet there is the possibility that due to a network error that the packet we inserted hidden data to could get lost. Although normal network communications will continue, due to the TCP timeout retransmission, the packet containing hidden data will not be resent. Therefore it is possible that some steganography data may not be received by the end host.

- **Man in the Middle attack** - a man in the middle attack occurs when data is intercepted by a third party and modified in order to gain or modify information[13]. The tool is susceptible to these kinds of attacks. This can occur if someone was to intercept the corrupt packets, and with knowledge of the algorithm, modify the lower byte of the corrupt checksum to change the data that is hidden within it. If the third party also knew of the senders IP address, the port intended for communication and the key then they could generate their own packets containing a spoofed IP address and send corrupt

packets in order for the receiver to receive information that was not intended.

2.8 Future Work

The tool has a large potential for future work. We can begin by addressing the weaknesses that were presented in the above section. One of the first problems that should be examined is the insertion of the key and data. This has the weakness that should the corrupt packets be identified as coming from this tool then decoding the data is relatively easy. This area could be improved by combining the key and data in some way. Possible approaches could use block ciphers to essentially encrypt the data using the key and then insert this value into the checksum. This would be beneficial as not only would it encrypt the data but the checksum would appear different for every corrupt packet sent. This would make detecting and decoding the steganography much more complex, and it has the effect of making man in the middle attacks harder.

Another problem that requires further study is the issue of lost packets that contain steganography. Further research should be conducted into this area as it is highly desirable to ensure delivery of steganographic packets. One possible approach is to either create the ability to acknowledge steganographic packets themselves. This could possible be achieved by the receiving host acknowledging with a corrupt packet itself. This way the receiving host can inform the sender, using a steganographic technique itself, that it received the hidden data.

The tool could also be expanded by allowing multiple hosts to send data. This means that a number of hosts could be used to send data to the receiver and software on the receiving host places the various packets of data in to the correct order. If this technique was introduced then the complexity of detecting it would be greatly increased.

2.9 Conclusion

In this paper we have presented a novel technique for network steganography. We developed this technique by purposely corrupting TCP packets. The innate behavior of TCP means that data is retransmitted when the receiving host does not receive the data correctly. By sending a corrupt packet we make the receiving host drop the packet and signal for it to be retransmitted. However, the checksum that we used contains hidden data, and by using software on the receiving host we are able to extract the data and present it to the user.

We have shown that the tool has practical use both by implementing it on a common platform and evaluating its advantages and disadvantages. This presentation allows law enforcement to be aware that the tool could be used in practice to communicate discreetly and thus contain criminal activities. This paper also highlights the fact that regular network activities, such as handling corrupt packets, can be harnessed and used in a malicious manner.

List of References

[1] D. Llamas, C. Allison, and A. Miller, "Covert channels in internet protocols: A survey," in *Proceedings of the 6th Annual Postgraduate Symposium about the Convergence of Telecommunications, Networking and Broadcasting*, 2005.

[2] I. S. Institute, "RFC 793," 1981, edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html. [Online]. Available: http://rfc.sunsite.dk/rfc/rfc793.html

[3] J. F. Kurose and K. W. Ross, *Computer Networking A Top-Down Approach*, 4th ed. Pearson Education, 2008.

[4] T. G. Handel and M. T. Sandford, II, "Hiding data in the osi network model," in *Proceedings of the First International Workshop on Information Hiding.* London, UK: Springer-Verlag, 1996, pp. 23–38.

[5] M. Wolf, "Covert channels in lan protocols," in *LANSEC '89: Proceedings on the Workshop for European Institute for System Security on Local Area Network Security.* London, UK: Springer-Verlag, 1989, pp. 91–101.

[6] C. Rowland, "Covert channels in the tcp/ip protocol suite," *First Monday*, vol. 2, no. 5, 1997.

[7] S. J. Murdoch and S. Lewis, "Embedding covert channels into tcp/ip." in *Information Hiding*, ser. Lecture Notes in Computer Science, M. Barni, J. H. Joancomartí, S. Katzenbeisser, and F. P. González, Eds., vol. 3727. Springer, 2005, pp. 247–261.

[8] J. G. N, R. Greenstadt, P. Litwack, and R. Tibbetts, "Covert messaging through tcp timestamps," in *Workshop on Privacy Enhancing Technologies*, 2002, pp. 194–208.

[9] K. Szczypiorski, "Hiccups: Hidden communication system for corrupted networks," in *Proc. of: The Tenth International Multi-Conference on Advanced Computer Systems ACS'2003, October 22-24, 2003*, 2003, pp. 31–40.

[10] F. Fuentes and D. C. Kar, "Ethereal vs. tcpdump: a comparative study on packet sniffing tools for educational purpose," *J. Comput. Small Coll.*, vol. 20, no. 4, pp. 169–176, 2005.

[11] J. Rutkowska, "The implementation of passive covert channels in the linux kernel," in *Chaos Communication Congress*, December 2004.

[12] C. Benvenuti.

[13] T. H. Barr, *Invitation to Cryptology.* Prentice Hall, 2001.

APPENDIX

TCP Steganography Tool Source Code

All source code is also avaliable at `http://cs.uri.edu/~nbennett/tcp/`

A.1 TCPStegSend.c

```
// Module
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>

// Packet handling
#include <linux/netdevice.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/in.h> // for IP proto numbers
#include <net/ip.h> // for ip_send_check

#include <linux/ethtool.h> // to turn off tx csum

// proc
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

__u32 in_aton(const char *);


static char *ip;
static int port;
static unsigned int key;
static char *stegData;
static unsigned int magic_ip;

module_param(ip, charp, 0);
module_param(port, int, 0);
module_param(key, int, 0);

struct packet_type stegSend;
static int stegSent;
```

```
static struct proc_dir_entry *steg_proc_dir;
static struct proc_dir_entry *steg_proc_input;
static int init_write = 1;

/*
 * Packet Type Handler
 */
int sendSteg(struct sk_buff *skb, struct net_device *dv,
                                      struct packet_type *pt){
   struct iphdr *iph;
   struct tcphdr *tcph;
   unsigned short randNum;

   unsigned long flags;
   local_irq_save(flags);
   local_irq_disable();

   if(skb->protocol == htons(ETH_P_IP))
           iph = ip_hdr(skb);
   else
           goto end;

   if(!stegSent && skb->pkt_type == PACKET_OUTGOING
      && iph->daddr == magic_ip
      && iph->protocol == IPPROTO_TCP){

      // fix up pointers
      skb->transport_header = skb->network_header + iph->ihl*4;
      tcph = tcp_hdr(skb);

      // check that packet is to the port we want
      // also that its not a syn packet
      if(tcph->dest == htons(port) && tcph->syn != 1){

              // packet is going where we want it
              // randomly select it for corruption
              get_random_bytes(&randNum, 1);
              if(randNum % 10 == 1){
                      // we selected the packet so now
                      // embed a byte of data in to the
                      // checksum
                      tcph->check = (key<<8) | stegData[0];
                      stegData = stegData + sizeof(char);
                      if(strlen(stegData) == 0){
```

```
                              stegSent = 1;
                              kfree(stegData);
                              stegData = NULL;
                      }
              }
      }
   }

end:
   kfree_skb(skb);
   return 0;
}

/*
 * Proc Read/Write functions
 */
int tcpsteg_proc_write(struct file *file, const char *buf,
                                unsigned long count, void *data){
    if(!stegSent && !init_write){
            return -EFAULT;
    }

    stegData = kmalloc(count+1, GFP_ATOMIC);
    if(!stegData){
            printk("Cannot allocate mem to stegData\n");
            return -ENOMEM;
    }

    memset(stegData, 0, count+1);
    copy_from_user(stegData, buf, count);
    stegData[count] = '\0';

    stegSent = 0;
    init_write = 0;

    return count;
}

int tcpsteg_proc_read(char *buf, char **start, off_t offset,
                               int count, int *eof, void *data){
    int len;
    if(stegSent || init_write)
            len = sprintf(buf, "All data sent\n");
    else
```

```
            len = sprintf(buf, "%s\n", stegData);

    return len;
}

/*
 * Module Load and Unload Functions
 */
int init_module(void){
   struct net_device *dev;

   if (!ip || !port || !key)
           return -1;

   magic_ip = in_aton(ip);
   stegSent = 1;

   // turn off tx csum'ing on eth0
   dev = dev_get_by_name(&init_net, "eth0");
   if(dev)
           ethtool_op_set_tx_csum(dev, 0);
   else{
           printk("couldnt get net device!\n");
           return -1;
   }

   // create proc entry for reading
   steg_proc_dir = proc_mkdir("tcpsteg", NULL);
   if(!steg_proc_dir){
           printk("couldn't create proc\n");
           return -1;
   }
   steg_proc_input = create_proc_entry("input", 0, steg_proc_dir);
   if(!steg_proc_input){
           printk("couldn't create proc\n");
           return -1;
   }
   steg_proc_input->write_proc = tcpsteg_proc_write;
   steg_proc_input->read_proc = tcpsteg_proc_read;

   stegSend.type = htons(ETH_P_ALL);
   stegSend.func = sendSteg;
   dev_add_pack(&stegSend);
```

```
    return 0;
}

void cleanup_module(void){
    if(!stegData)
            kfree(stegData);

    dev_remove_pack(&stegSend);
    remove_proc_entry("input", steg_proc_dir);
    remove_proc_entry("tcpsteg", NULL);
    printk("TCPStegSend unloaded\n");
}

/*
 * Utility Code
 * The below code taken from Phrack Issue 55
 */
__u32 in_aton(const char *str) {
    static unsigned long l;
    static unsigned int val;
    static int i;

    l = 0;
    for (i = 0; i < 4; i++) {
            l <<= 8;
            if (*str != '\0') {
                    val = 0;
                    while (*str != '\0' && *str != '.') {
                            val *= 10;
                            val += *str - '0';
                            str++;
                    }
                    l |= val;
                    if (*str != '\0')
                            str++;
            }
    }
    return(htonl(l));
}
```

A.2 TCPStegRecv.c

```
// Module
#include <linux/kernel.h>
```

```
#include <linux/module.h>

// Packet handling
#include <linux/netdevice.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/in.h> // for IP proto numbers

#include <linux/proc_fs.h>

#define STEG_BUF_SIZE 512

__u32 in_aton(const char *);

static char *ip;
static int port;
static unsigned int key;
static unsigned int magic_ip;

module_param(ip, charp, 0);
module_param(port, int, 0);
module_param(key, int, 0);

struct packet_type stegRecv;

static char *stegData;
static int buf_len;
static struct proc_dir_entry *steg_proc_dir;
static struct proc_dir_entry *steg_proc_output;

/*
 * Packet Type Handler
 */
int receiveSteg(struct sk_buff *skb, struct device *dv,
                                      struct packet_type *pt){
   static unsigned int doff, data_len;
   static int csum, rcvd_csum;
   static struct iphdr *iph;
   static struct tcphdr *tcph;
   int cur_buf_len;
   char *new_buf;

   iph = ip_hdr(skb);
```

```
if (iph->protocol == IPPROTO_TCP){
   // set the transport header then get the tcp header
   skb_set_transport_header(skb, iph->ihl*4);
   tcph = tcp_hdr(skb);

   // calc offset into data field of TCP packet
   doff = tcph->doff*4;
   // length of actual data in TCP packet
   data_len = ntohs(iph->tot_len) - doff - iph->ihl*4;

   // calc the checksum for the data only
   csum = csum_partial(skb->transport_header + doff,
                                data_len, 0);

   if (tcph->dest == htons(port)){
       // could be steg packet, is checksum invalid?
       // set tcp checksum to 0
       rcvd_csum = tcph->check;
       tcph->check = 0;

       // form pseudo header and calc checksum
       csum = csum_tcpudp_magic(iph->saddr,
                       iph->daddr,
                       data_len + doff,
                       iph->protocol,
                       csum_partial(skb->transport_header,
                                       doff, csum)
                       );

       if((csum != rcvd_csum) &&
                       (key<<8) == (rcvd_csum & 0xff00))
       {
               // Check if we need to allocate more mem
               cur_buf_len = strlen(stegData);
               if(cur_buf_len+1 == buf_len){
                       new_buf = kmalloc(
                           (buf_len + STEG_BUF_SIZE) *
                           sizeof(char),
                           GFP_ATOMIC
                           );
                       if(!new_buf){
                            printk("error allocating mem\n");
                            goto end;
```

```
                                }
                                memset(new_buf,
                                       0,
                                       (buf_len+STEG_BUF_SIZE) *
                                       sizeof(char));

                                memcpy(stegData, new_buf, buf_len);
                                kfree(stegData);
                                stegData = new_buf;
                                buf_len = buf_len + STEG_BUF_SIZE;
                        }
                        stegData[cur_buf_len] = rcvd_csum & 0x00ff;
                        printk("cur_buf_len =  %i\n", cur_buf_len);
                        printk("stegData = %s\n", stegData);
                }


                }
                // either way we need to put the packet back to
                // what it was and let the actual TCP layer handle it
                tcph->check = rcvd_csum;
            }
        }

end:
        kfree_skb(skb);
        return 0;
}

/*
 * Proc Read/Write functions
 */
int tcpsteg_proc_read(char *buf, char **start, off_t offset,
                                int count, int *eof, void *data){
        int len;
        int data_len = strlen(stegData);

        if(offset != 0)
                return 0;

        if(data_len == 0)
                len = sprintf(buf, "No data to output\n");

        else{
```

```
        memset(buf, 0, count);
        len = snprintf(buf, data_len+1, "%s\n", stegData);
        memset(stegData, 0, buf_len*sizeof(char));
    }

    return len;
}

/*
 * Module functions
 */
int init_module(void){
    if (!ip || !port || !key)
        return -1;

    magic_ip = in_aton(ip);

    stegData = kmalloc(STEG_BUF_SIZE * sizeof(char), GFP_ATOMIC);
    buf_len = STEG_BUF_SIZE;
    if(!stegData){
        printk("Error allocating memory to buffer\n");
        return -1;
    }
    memset(stegData, 0, STEG_BUF_SIZE*sizeof(char));

    // create proc entry for reading
    steg_proc_dir = proc_mkdir("tcpsteg", NULL);
    if(!steg_proc_dir){
        printk("couldn't create proc\n");
        return -1;
    }
    steg_proc_output = create_proc_entry("output",
                                         0,
                                         steg_proc_dir);
    if(!steg_proc_output){
        printk("couldn't create proc\n");
        return -1;
    }
    steg_proc_output->read_proc = tcpsteg_proc_read;

    stegRecv.type = htons(ETH_P_IP);
    stegRecv.func = receiveSteg;
    dev_add_pack(&stegRecv);
```

```
   return 0;
}

void cleanup_module(void){
   kfree(stegData);
   dev_remove_pack(&stegRecv);
   printk("TCPStegRecv unloaded\n");
}

/*
 * Utility functions
 */
__u32 in_aton(const char *str) {
  static unsigned long l;
  static unsigned int val;
  static int i;

  l = 0;
  for (i = 0; i < 4; i++) {
          l <<= 8;
          if (*str != '\0') {
                  val = 0;
                  while (*str != '\0' && *str != '.') {
                          val *= 10;
                          val += *str - '0';
                          str++;
                  }
                  l |= val;
                  if (*str != '\0')
                          str++;
          }
  }
  return(htonl(l));
}
```

BIBLIOGRAPHY

Barr, T. H., *Invitation to Cryptology.* Prentice Hall, 2001.

Benvenuti, C.

Farid, H., "Detecting steganographic messages in digital images," Dartmouth College, Hanover, NH, USA, Tech. Rep., 2001.

Fridrich, J. J., "Feature-based steganalysis for jpeg images and its implications for future design of steganographic schemes," in *Information Hiding*, 2004, pp. 67–81.

Fuentes, F. and Kar, D. C., "Ethereal vs. tcpdump: a comparative study on packet sniffing tools for educational purpose," *J. Comput. Small Coll.*, vol. 20, no. 4, pp. 169–176, 2005.

Handel, T. G. and Sandford, II, M. T., "Hiding data in the osi network model," in *Proceedings of the First International Workshop on Information Hiding.* London, UK: Springer-Verlag, 1996, pp. 23–38.

Hetzl, S., "Steghide."

Institute, I. S., "RFC 793," 1981, edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html. [Online]. Available: http://rfc.sunsite.dk/rfc/rfc793.html

(JPEG), J. P. E. G., *JPEG ISO/IEC 10918-1 ITU-T Recommendation T.81.*

Kessler, G., "Steganography: Hiding data within data," September 2001. [Online]. Available: http://www.garykessler.net/library/steganography.html

Kurose, J. F. and Ross, K. W., *Computer Networking A Top-Down Approach*, 4th ed. Pearson Education, 2008.

Llamas, D., Allison, C., and Miller, A., "Covert channels in internet protocols: A survey," in *Proceedings of the 6th Annual Postgraduate Symposium about the Convergence of Telecommunications, Networking and Broadcasting*, 2005.

Menon, R. J., "MP3 Steganography and Steganalysis," Master's thesis, University of Rhode Island, 2009.

Murdoch, S. J. and Lewis, S., "Embedding covert channels into tcp/ip." in *Information Hiding*, ser. Lecture Notes in Computer Science, Barni, M., Joancomartí, J. H., Katzenbeisser, S., and González, F. P., Eds., vol. 3727. Springer, 2005, pp. 247–261.

N, J. G., Greenstadt, R., Litwack, P., and Tibbetts, R., "Covert messaging through tcp timestamps," in *Workshop on Privacy Enhancing Technologies*, 2002, pp. 194–208.

Pevný, T. and Fridrich, J., "Merging Markov and DCT features for multi-class JPEG steganalysis," in *Proceedings SPIE, Electronic Imaging, Security, Steganography, and Watermarking of Multimedia Contents IX, San Jose, CA, January 29–February 1*, Delp, E. and Wong, P., Eds., vol. 6505, January 2007, pp. 03–14.

Provos, N. and Honeyman, P., "Hide and seek: An introduction to steganography," *IEEE security & privacy*, vol. 1, no. 3, pp. 32–44, 2003.

Rowland, C., "Covert channels in the tcp/ip protocol suite," *First Monday*, vol. 2, no. 5, 1997.

Rutkowska, J., "The implementation of passive covert channels in the linux kernel," in *Chaos Communication Congress*, December 2004.

Sallee, P., "Model-based steganography," *Lecture Notes in Computer Science*, pp. 154–167, 2003.

Shi, Y. Q., Chen, C., and Chen, W., "A markov process based approach to effective attacking jpeg steganography." in *Information Hiding*, ser. Lecture Notes in Computer Science, Camenisch, J., Collberg, C. S., Johnson, N. F., and Sallee, P., Eds., vol. 4437. Springer, 2006, pp. 249–264.

Szczypiorski, K., "Hiccups: Hidden communication system for corrupted networks," in *Proc. of: The Tenth International Multi-Conference on Advanced Computer Systems ACS'2003, October 22-24, 2003*, 2003, pp. 31–40.

Wallace, G. K., "The jpeg still picture compression standard," *Commun. ACM*, vol. 34, no. 4, pp. 30–44, 1991.

Westfeld, A. and Pfitzmann, A., "High capacity despite better steganalysis (F5–a steganographic algorithm)," in *Information Hiding, 4th International Workshop*, vol. 2137, 2001, pp. 289–302.

Wolf, M., "Covert channels in lan protocols," in *LANSEC '89: Proceedings on the Workshop for European Institute for System Security on Local Area Network Security.* London, UK: Springer-Verlag, 1989, pp. 91–101.