MODERN STEGANOGRAPHY: AN OVERVIEW

By

Craig Stephen Miles

Bachelor of Science, Mathematical Sciences, Oregon State University, Corvallis, OR 2007

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
In Computer Science

The University of Montana
Missoula, MT

Spring 2010

Approved by:

Perry Brown, Associate Provost for Graduate Education
Graduate School

Dr. Joel Henry, Chair
Computer Science

Dr. Min Chen
Computer Science

Dr. George McRae
Mathematical Sciences

Miles, Craig, M.S., Spring 2010 *Computer Science*

Steganography: A Survey of Hiding Data in Common File Types

Chairperson: Dr. Joel Henry

Steganography is the art and science of writing hidden messages in such a way that no one, apart from the sender and intended recipient, suspects the existence of the message. While originally limited to using objects from the physical world as a channel for covert communication, the omnipresence of the modern PC has provided a new source of cover-objects for steganographic communication: digital files.

In digital steganography, hidden messages are embedded into computer files in such a way that the message's existence is (hopefully) undetectable by anyone except the sender and the recipient. To an outside observer, the object containing the embedded message should appear and function exactly as it would if it did not contain such a message.

To illustrate the capacity of modern steganography to provide a medium for secret communication, algorithms are developed herein to embed and recover messages within digital images. The same algorithms are subsequently implemented in an accompanying computer program: CommonSteg.

TABLE OF CONTENTS

TABLE OF FIGURES

ACKNOWLEDGMENTS

I wish to convey my sincerest gratitude to the faculty of the Computer Science department at The University of Montana, and particularly to Professor Joel Henry, for providing me the opportunity to continue my academic career. I'll never be able to fully express how profoundly thankful I am that you saw potential in me.

CHAPTER 1 INTRODUCTION

1.1 Purpose

The purpose of this thesis is to demonstrate a mastery of available scholarship in the field of Digital Steganography. Briefly, steganography is the study of sending invisible messages; messages which are hidden within some other innocuous object. Following an introduction and formalization of the field, a survey of the various way in which it may be applied to common computer file types, with a specific focus on digital images, is conducted.

A computer program called CommonSteg has been written in conjunction with this thesis to demonstrate many of methods of hiding data within common file types as described herein.

The types of files for which steganographic techniques will be described include JPEG, GIF, PNG images, and ZIP compressed archives. In addition to steganographically hiding information in these specific file types, the concept of hiding information at higher levels of abstraction, such as in image data itself regardless of the container type, will also be explored.

1.2 What is Steganography?

Steganography, a rough translation of 'secret writing' from Greek, is traditionally defined as the art and science of writing hidden messages in such a way that no one,

apart from the sender and intended recipient, suspects the existence of the message. The first recorded use of the term was in *Steganographia*, a primer on early cryptography and steganography written by Johannes Trithemius in 1499. The full title of Trithemius' treatise roughly translated from Latin to English is: "Steganography: the art through which writing is hidden requiring recovery by the minds of men. (Judge 2001)" *Steganographia* was written as a trilogy; the first two often being described as some of the earliest books on cryptology. The third part, however, is ostensibly a book on occult astrology in which Trithemius included tables of numbers, Latin characters, and zodiac symbols. It wasn't until nearly five centuries later that researchers discovered that Trithemius had hidden additional messages in the book, including the humorous note, "The bearer of this letter is a rogue and a thief. Guard yourself against him. He wants to do something to you." In hiding these messages, Trithemius managed to steganographically hide information within the series of books in which he had formalized the concept in the first place.

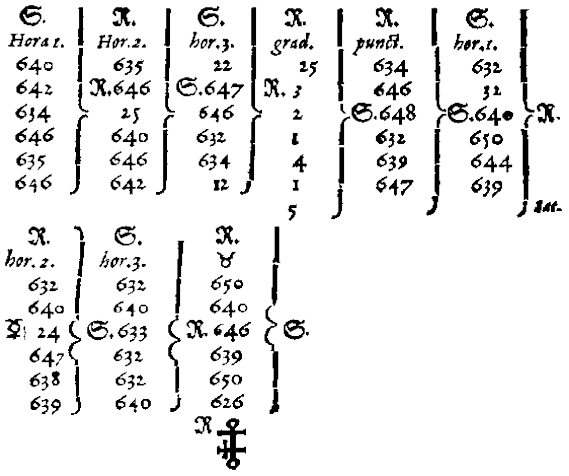


Figure 1: Numeric table from Book 3 of Trithemius' "Steganographia" (Trithemius 1621)

A common description of steganography is Simmons' "Prisoners' Problem" (Simmons 1984), summarized and modernized by (Cachin 2005): Alice and Bob are in jail, locked up in separate cells far apart from each other and wish to devise an escape plan. They are allowed to communicate by means of sending messages via the warden Eve's trusted couriers, provided they do not deal with escape plans. If Eve detects any sign of conspiracy, she will thwart the escape plans by transferring both prisoners to high-security cells from which nobody has ever escaped. Alice and Bob succeed if they can exchange information allowing them to coordinate their escape and Eve does not become suspicious.

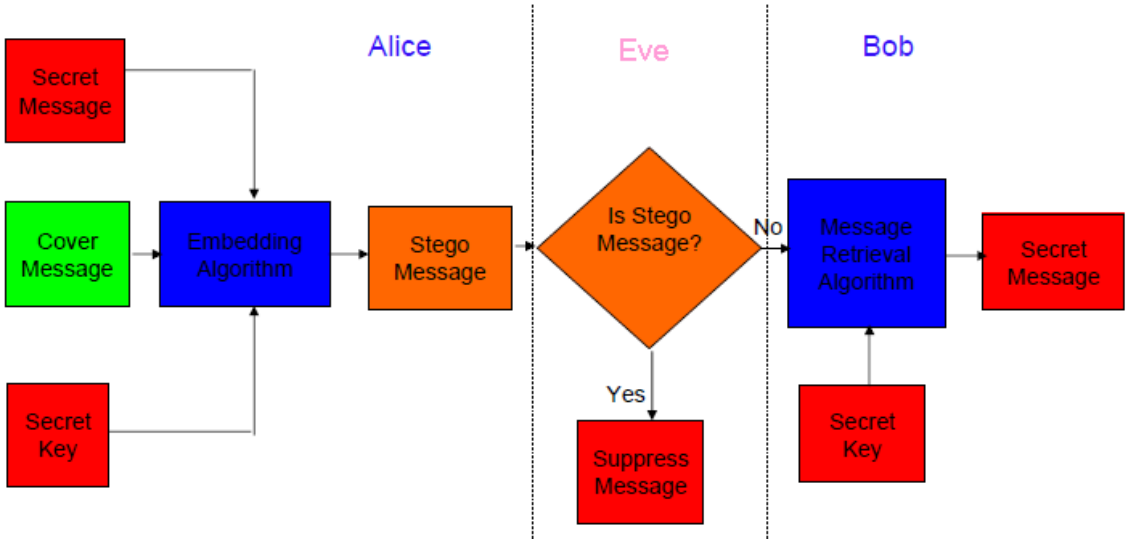


Figure 2: Illustration of Simmons' modernized "Prisoners' Problem" (Unknown 2007)

Throughout history there have been numerous documented accounts of the passing of concealed messages, even before Trithemius formalized the idea of Steganography. One of the earliest examples is that of Histaiaeus in the 5th century BC. He tattooed a

message onto the shaved head of a messenger who upon regrowing his hair delivered himself to the recipient. Another example from antiquity involved the sending of a message to the Spartans warning of a Persian invasion. Messages of the time were often written into wax covered wooden tablets. In order to covertly deliver the warning, the Greeks scraped the wax off of a tablet, carved the warning into the wood itself, and then reapplied a layer of wax thereby giving the impression that the tablets were indeed blank.

More recently, Nazi Germany during World War II employed the usage of the Microdot to serve as a Steganographic means of communication. No larger than a period or the tittle of an *i* or *j*, what appeared to be merely a dot to the naked eye was actually an image or text that had been shrunken down to a miniscule size, visible only via the use of a microscope.

During the Vietnam War, American Admiral (then Commander) Jeremiah Denton Jr. was captured by the North Vietnamese. He was forced by his captors to take part in a television interview during which he blinked in Morse code the word "T-O-R-T-U-R-E" in order to let it be known that the Vietnamese were torturing him and the other prisoners of war. In doing so, Commander Denton successfully managed to send a Steganographic message via the enemy's own communication channels unbeknownst to them; a truly remarkable feat.

The majority of documented cases regarding the usage of Steganographic techniques prior to the personal computer age seem to be primarily related to military endeavors; however that is no longer the case in modern times. There is often a need for privacy in communication, however many people living under repressive regimes and those who work for many companies may find that the usage of encryption is banned by those in charge. For example, a 2000 survey (Madsen and Banisar 2000) found that Belarus, Burma, China, Kazakhstan, Pakistan, Russian, Tunisia, and Vietnam all had placed strong domestic controls on the use of cryptography. For those who are in a situation requiring privacy of communication yet are unable to send encrypted messages in the open without fear of reprisals, steganography can provide a safer means of getting one's message out. Steganographic techniques have been used by dissidents and human rights campaigners around the world to smuggle information and messages out of repressive areas, and the same techniques have also been used by corporate whistleblowers to leak information.

Various forms of Steganography have indeed been used by numerous individuals, organizations, and governments over the centuries for a multitude of purposes, including military, diplomatic, and personal communications, as well as for the concealment of intellectual property (Judge 2001).

Many of the earliest forms of steganography were no more complex than the usage of invisible ink to write additional text between the lines, however through the passage of

time, steganographic models have become increasingly complex. In the modern age, steganography most often refers to the hiding of a message or information within a computer file. For example, messages or information can be hidden within an image file or music file without impeding the ability of that file to perform as a user unaware of the secret message's existence would expect.

1.3 Formalization

Before proceeding into a discussion of how information may be hidden into common computer files, it is first useful to formalize Steganography by defining its related terminology.

A cover-<data_type>, where <data_type> represents the medium that will carry the hidden message, is the seemingly innocuous object into which the message will be placed. For example, if a message is to be concealed within an image, the original image is to be referred to as a cover-image. When the cover-<data_type> is combined steganographically with the message to be hidden, the resulting output is referred to as the stego-<data_type>. In the previous example, the output of steganographically hiding a message into the cover-image is a stego-image. The process by which the message is hidden into the cover-<data_type> is referred to as the stego-system, or more specifically, the embedding function of the stego-system. If some sort of additional information beyond the stego-<data_type> itself is required in order to recover the hidden message, that information shall be referred to as the stego-key. An

individual who attempts to determine whether or not a given message contains steganographically hidden information is called the stego-analyst. This information hiding terminology was agreed upon in (Pfitzmann 1996). A stego-system is considered to be secure if a stego-analyst cannot distinguish between a message containing no hidden information and a stego-message (Cachin 2005).

The model in Figure 3 is an illustrated version of the basic model of a steganographic system as defined using the terminology of (Pfitzmann 1996).

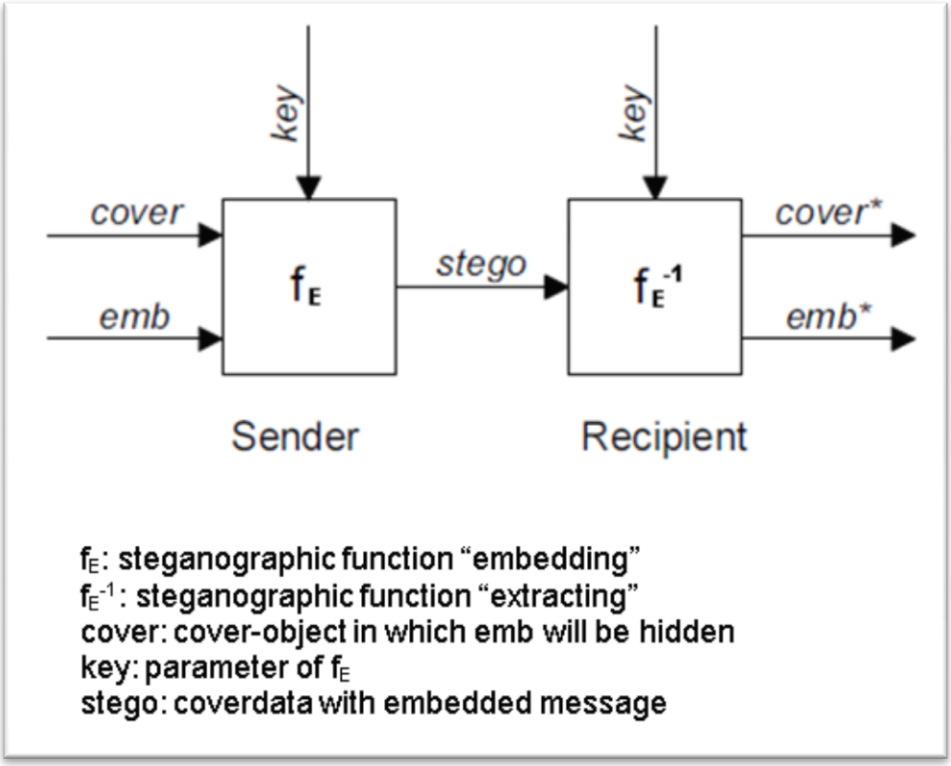


Figure 3: Basic Model of a Steganographic System (Zöllner, et al. 1998)

The input, *cover*, is the innocuous data into which *emb* will be hidden by the function f_E. If necessary to the particular stego-system, f_E will also make use of the shared-secret

input: *key*. f_E outputs *stego*, data which appears to be the same as *cover*, yet also contains *emb*. The function f_E^{-1} allows the recipient to extract emb^* and possibly $cover^*$ from *stego*, both of which should be equal to or an approximation of the original *emb* and *cover* respectively.

Using Figure 3 as guidance, we can define the embedding process as follows:

$$stego = f_E(cover, emb, key).$$

This assumes an *m* bit cover into which we hide an *n* bit *emb*. For later purposes, I also adopt the following notation:

C: the set of all bitstrings
cover: actual bitstring of length m (cover ∈ C)
E: the set of all bitstrings
emb: actual bitstring of length n (emb ∈ E)
K the set of all keys
key: actual key (key ∈ K)
S: the set of all bitstrings (S = C)
stego: actual stego, i.e. bitstring the contains emb (stego ∈ S)

CHAPTER 2 STEGANOGRAPHIC SECURITY

2.1 Overview

Steganography and cryptography serve similar purposes. Both fields attempt to provide private communication between multiple parties, however they go about it by different means. In a cryptographic system, all parties are aware of the existence of the encrypted messages. The cryptographic system is said to be broken if someone besides the intended recipient is able to obtain any information about the original message. As opposed to cryptography, a steganographic system fails if anyone but the sender and intended recipient even becomes aware of the existence of the hidden message. As such, a steganographic system's security can be defined in terms of *undetectability*. A perfectly secure stego-system would be undetectable 100% of the time.

Referring back to the Prisoners' Problem, steganalysis is the set of techniques by which Eve may attempt to distinguish between innocent messages and stego-messages. Note that this distinction may need to be made without Eve knowing either the stego-key (if there is one) or the stego-system employed. Eve need not, however, ascertain the actual steganographically hidden message in order to succeed, but rather she must only determine the existence of the message.

Breaking a steganographic system thus has two stages (Zöllner, et al. 1998):

1 The attacker can detect that steganography has been used.

2 Additionally, he may be able to read the embedded message.

Multiple approaches have been developed in order to define and quantify steganographic security. In (Zöllner, et al. 1998), the authors provided an analysis to show that information theoretically secure steganography is possible if both the embedding operation is non-deterministic and *emb* is independent from the *cover* and the *stego*. Whereas Zöllner's methodology is useful for ascertaining whether a scheme enjoys perfect security from a theoretical viewpoint, (Katzenbeisser and Petitcolas 2002) describe a model more applicable for characterizing the security of real world steganographic systems. Summaries of both of these models for describing steganographic security follow.

2.2 Zöllner et al.'s Information Theoretic Security Model

In the field of cryptography, multiple types of crypto-analytical attacks are defined, all being based upon how much information the attacker is presented with. In a ciphertext-only attack (also known as a known ciphertext attack), the attacker is assumed to have access only to a set of ciphertexts. The attack is considered successful if the attacker is able to ascertain the plaintext(s). There is also the notion of a chosen-plaintext attack. In such a scenario, the attacker has the ability to choose arbitrary plaintexts to be encrypted and obtain the corresponding ciphertexts (Wikipedia contributors 2010). A chosen-plaintext attack is successful if the attacker is able to deduce information related to the cryptographic scheme in use such that its security is reduced.

Both of these attacks can be morphed into similarly structured attack models for the purpose of steganalysis. The ciphertext-only attack becomes the stego-only attack, in which the attacker only knows *stego*. The chosen-plaintext attack becomes a stego-cover attack, where the attacker knows both *stego* and *cover*. Using information theory, we can prove that secure steganography is impossible in a stego-cover attack if the stego-system is deterministic. By deterministic, it is meant that given the same inputs to the stego-system, the output will always be the same.

Information theory is a statistical theory dealing with the limits and efficiency of information processing (Princeton University n.d.). A standard measure of the theory is *entropy*. Entropy quantifies the uncertainty when encountering a random variable (Wikipedia contributors 2010). For a given alphabet X, the entropy H(X) describes the "uncertainty about X", which more accurately means the uncertainty about the occurrence of a certain element $x \in X$. The remaining uncertainty about X when knowing Y is defined as $H(X|Y)$. $H(X,Y) = H(X) + H(X|Y)$ is the "union" of both entropies, also known as the joint entropy. Finally, the mutual information $I(X;Y) = H(X) – H(X|Y)$ describes the amount of information about X you get if you know Y (Gallager 1968).

Zöllner et al state that a stego-system is information theoretically secure if the attacker cannot gain any information about *emb* or *E* by examining *stego* and *cover*, thus there must be zero mutual information:

$$I(E;(S, C)) = H(E) – H(E|(S, C)) = 0 \quad \textbf{(1)}$$

This reduces to the fundamental security condition:

$$H(E|(S, C)) = H(E) \quad \textbf{(2)}$$

Plainly written, this states that knowledge of *stego or cover* may not reduce the uncertainty in *emb*. The result of this observation is that *emb* (or E) must always be **independent** of *stego* and *cover* (or S and C respectively).

Because an attacker should not be able to differentiate between a cover and a stego, not only can we assume that their alphabets S and C are the same, but also that their entropies H(S) and H(C) are equal. We do, however, see differences in the conditional entropies between the cases of whether or not an *emb* was embedded:

- Without embedded information: $H(S|C) = H(C|S) = 0$
- With embedded information: $H(S|C) = H(C|S) > 0$

Via the connection of entropy and information, we see that the uncertainty about S if we know C (or vice versa) corresponds to the information about E that you can get by looking at S and C. Therefore, by embedding $emb \in E$ into $cover \in C$, mutual information is inevitably non-zero:

$$I(E;(S,C)) = H(E) – H(E|(S,C)) > 0 \quad (3)$$

Thus:

$$H(E|(S,C)) < H(E) \quad (4)$$

This result, however, contradicts the security condition of (2). As such, we find that the necessary and sufficient condition for secure steganography is:

$$H(S|C) = H(C|S) = 0 \quad (5)$$

As such, for a stego-system to be secure, it must hold that:

$\forall i \in N, stego_i \in S, cover_i \in C : stego_i = cover_i$.

The secure steganography is reduced to a practically irrelevant case: *cover* ≡ *stego*. In other words, in a deterministic stego-system, the only way to provide perfect security is to find a *cover* that already contains *emb*. If we disregard this implausible case, we find that the security condition in (2) can never be fulfilled if f_E is deterministic and the attacker knows both *cover* and *stego*.

In order to remedy this lack of security, Zöllner et al propose an alternative to the basic steganographic model from Figure 3. In this new model, the *cover* is selected from a defined set of possible covers called the *source*. Rather than allowing the attacker to know the actual cover, the attacker may only know the source from which the cover was derived. For example, consider that the *cover* is created from a sampling of an analog input, i.e. line audio. Even if the attacker knows that said analog input is the *source* of the *cover*, as long as the produced *stego* remains within the domain of possible analog input, the manipulations added during the embedding process cannot be recognized as such. If we define the *source* of *cover* to be C_S, then the previous statement holds true as long as the following conditions are true:

(a) $H(C,C_S) \geq H(C) + H(E)$

(b) $H(C|S) \geq H(E)$

(c) $H(C_S|S) \geq H(E)$

In such a scenario, as long as the attacker does not know the actual cover, but rather just the source from which the cover is derived, then the steganographic system is proven to be information theoretically secure.

2.3 Katzenbeisser and Petitcolas' Practical Model

The previous information theoretic model for steganographic security assumes a steganalyst equipped with unlimited computational power and a detailed knowledge of statistical information of the cover-object's source. While holding these assumptions is a necessity for the design of a theoretically perfectly secure steganographic system, in practice they are rarely if ever met. In (Katzenbeisser and Petitcolas 2002), Katzenbeisser and Petitcolas attempt to define steganographic security in a more relevant and practical way.

The drawbacks to using an information theoretical model to define steganographic security are multifold. First, while it is possible to come up with steganographic systems fulfilling the previous definitions of perfect steganographic security, it turns out that they are all impractical in most situations. The proposed systems are generally some variant of the Vernam scheme from cryptography (one-time pad), redefined to fit under the previous security definitions. The same limitations that make the usage of the one-

time pad difficult, such as insecure key distribution and the requirement of perfect randomness for the creation of the key, carry over to the proposed stego-systems.

Additionally, the previous security definition requires knowledge of the probability distribution of *C* which is the set of all bitstrings from which the cover may be derived (also previously referred to as the *source*). In practice, this probability distribution is rarely known. For example, while it may be possible to approximately model the distribution of all "meaningful" grey-scale images, it is not possible to compute their exact distribution. If the distribution's approximation error is greater than the modification applied during the embedding process, then said approximation becomes irrelevant to the decision making process. Complicating the usage of a meaningful distribution approximation even more in the decision process, the steganalyst may need to find an approximate model for the messages that are "usually" sent between the sender and the recipient, as using a general model might not be sufficient. For example, consider the cover-source to be the set of grayscale images; however, the sender only ever uses a specific subset of the total possible shades of gray in his covers. In such a scenario, the stego-analyst's approximate model of all "meaningful" grayscale images is no longer relevant to the particular problem.

Finally, the previous model assumes that the stego-analyst has access to limitless computing power. That is an unreasonable assumption in the real world. In practice, one would generally be convinced of the security of a steganographic system if it were

to pass all probabilistic polynomial tests for determining whether or not a possible stego-object actually contained an embedded message.

Intuitively, the relative inability of an attacker to distinguish covers (containing no hidden information) from stego-objects is used to quantify the security of steganographic systems. In order to model this with regard to the previously discussed limitations of the information theoretical security models, Katzenbeisser and Petitcolas define a *steganographic decision problem* faced by the steganalyst: "Given any cover or stego-object, he must be able to guess (better than random) whether a secret message is actually contained in the object or not" (Katzenbeisser and Petitcolas 2002). In order to facilitate this decision, the attacker can compare the suspected stego-object against "common" objects usually sent between the sender and recipient. By comparing the stego-object against the set of all transmitted objects, the eavesdropper may reevaluate and improve the decision problem strategy.

In addition to the previously stated notation, we now define a steganographic system as a 3-tuple <G, E, D> of probabilistic polynomial time algorithms. The algorithm G serves as the key generation process, which outputs a random $k \in K$ to serve as the stego-key. Algorithm E is the embedding operation which upon inputs $c \in C$, $m \in M$, and k given by algorithm G, outputs a stego-object $s \in C$. Finally, D represents the Message Retrieval Algorithm, which when given inputs s and k outputs $m' \in \{0,1\}^*$ where if s actually contained a secret message m, then $m = m'$. Finally, the steganographic decision

problem is formally defined as follows: Given $s \in C$, determine if there exists a $k \in \{0,1\}^*$ in the range of G and a message $m \in M$ such that $D(s,k) = m$.

Given this decision problem, we can now model and quantify the security of the steganographic system. To do this, we assume the eavesdropper has access to two oracles. The first, called the *steganographic oracle* is an infinite sequence of covers c_1, c_2, …, such that each is a member of C. This oracle allows the observer to generate an arbitrarily large number of possible cover-objects. The *structure evaluation oracle*, on the other hand, allows the observer to generate stego-objects. It takes as input any $c \in C$ and any $m \in M$ and returns the corresponding stego-object s containing m. This *structure evaluation oracle* works as a black-box, embedding messages into covers given a fixed key k, even if k is unknown to the attacker.

The security test is then modeled as a game between the eavesdropper and a judge. First, the judge runs G in order to generate a $k \in K$. He uses this k to create the *structure evaluation oracle*, which is given to the eavesdropper along with the *steganographic oracle*. The eavesdropper is then allowed to query both oracles as many times as he likes, where he may perform polynomial-time computations on any objects he receives from the oracles. Once finished, the judge uses the *steganographic oracle* to generate two covers: $c_1, c_2 \in C$. He also selects a message $m \in M$ randomly and uses the *structure evaluation oracle* to embed m into c_2 resulting in s. Finally, using a fair coin toss to decide which, he either provides the eavesdropper c_1 or s. The eavesdropper then

performs whatever probabilistic tests he desires in order to determine whether or not he was given the cover-object c_1 or s containing some hidden message m and publishes his guess. The steganographic system is then secure for the *steganographic oracle* if the eavesdropper's advantage, that is the probability of a correct guess minus ½, is negligible. The ½ is subtracted because the eavesdropper can randomly guess correctly between c_1 and s half of the time.

A steganographic system is said to be U-Secure if for a given *steganographic oracle* U, at the end of the previously described game, the eavesdropper has a negligible advantage. The steganographic system S = <G, E, D> is then said to be conditionally secure if for a set of *steganographic oracles* C, ∀ U ∈ C, S is U-secure. In other words, the steganographic system is said to be conditionally secure if it is secure for every oracle. It is referred to as conditionally secure due to the fact that many schemes that would be considered such would not be considered perfectly secure from an information theoretic viewpoint, in which it is often assumed the eavesdropper has either unlimited computational power or access to an exact probability distribution in order to aid in the decision making process.

2.4 A Conditionally Secure Steganographic System

Having discussed steganographic security in the context of perfect security as defined by information theory and conditional security in terms of what is applicable to the real

world, a steganographic system fulfilling the conditions of conditional security yet remaining information theoretically insecure will now be outlined.

The creation of such a system, even with the new and more reasonable constraints on the eavesdropper, still faces several difficulties. Significant leeway must be given in what may be permissible to use as the cover source. The security of a steganographic system S can be based on some set of computational problems that are thought to be intractable. For example, assume a system for which there is an embedding function E that embeds some message m into the least significant bits of an image such that the distribution of the least significant bits remains unchanged. Let G, the stego-key generation algorithm of S, be the key generation function of the RSA public-key cryptosystem. If necessary, see Appendix A for an explanation of the RSA encryption system. Assume now that for every transmission of an image between the sender and the recipient, if there is no steganographic message to be hidden then the image is modified as follows: a random string x of length n is selected by the sender, a 0 is appended to the end, and then it is encrypted using RSA and the resulting ciphertext is embedded into the image. On the other hand, if there is to be a steganographic message hidden in the cover, the sender selects his message m, pads the end of m with bits until it reaches length n, then appends a 1. This concatenated message is then encrypted with RSA and embedded into the image. The end result of the employment of such a system is that all messages passed along the communication channel always contain some encrypted embedded string. In order then for an eavesdropper to decide

with non-negligible advantage that a communication contains a steganographic message rather than a random string, he must be able to determine the least significant bit of the embedded message. It was shown in (Goldwasser, Micali and Tong 1982) that if an attacker is successfully able to decrypt the least significant bit of an RSA encrypted ciphertext, then he would be able to efficiently decrypt the entire message. As such, in order for the eavesdropper to differentiate between covers containing steganographic messages and those containing random garbage, it would then be implied that he had invented an attack against RSA cryptography capable of ascertaining the least significant bit of the encrypted message, thereby violating the intractability of the entire cryptographic system.

Such a system would be considered conditionally secure because no polynomial time algorithm to determine the least significant bit of an RSA encrypted ciphertext is known to exist. However, the same system cannot be said to be perfectly secure in the information theoretic sense due to the fact that in that security model there are no restrictions on unbounded computational power. In the information theoretic models, the steganographic system is considered insecure even if the model is susceptible only to a brute-force attack.

As discussed before, the broad measure of a steganographic system's security is its undetectability. However, the very notion that the attacker should not be able to gain any information about the embedded message implies that there is indeed a message.

A similar contradiction exists in the steganographic system that has been laid out in this section. Kerkhoff's Principle is an axiom of cryptography which states that the security of a cryptographic system should remain uncompromised even if everything about the cryptosystem, except the key, is public knowledge (Katzenbeisser and Petitcolas 1999). The same idea holds for steganography. For true steganographic security, the entire security of a stego-system must rely on the stego-key alone. It must be assumed that an attacker knows exactly how the stego-system works. If a stego-system does not employ a stego-key, it is trivial for an attacker who understands the system to determine whether or not an embedded message is present, thereby compromising any security.

Referring back to the Prisoners' Problem once again, we must assume that by Kerckhoffs' Principle the warden is fully aware of how the system works. In such a scenario, there would be no reason for the warden to allow any cover messages from the source as described to be delivered between Alice and Bob. It remains an open problem to determine a provably secure steganographic system whose cover source would not inherently raise suspicion that steganographic communication is at least occasionally taking place.

2.5 Steganographic Capacity

The maximum size of a steganographically hidden message is limited by the number of redundant bits within the cover-object. Steganographic capacity in general is quantified as the maximum amount of information that can be embedded into a cover-object and

that then can be reliably recovered from the stego-object, under the constraints of undetectability and perceptual intactness (Kharrazi, Sencar and Memon 2004). Because steganographic systems have the fundamental requirement of undetectability, when data is hidden steganographically into a digital cover medium the embedding process must preserve not only the perceptual intactness of the cover-object but also its underlying statistical properties as well.

In the vast majority of cases, the smaller the embedded message is and the larger the cover is, the less probability there is of introducing detectable statistical changes. Each steganographic system seems to have its own upper-bound for its maximal safe message length that determines the number of message bits that can be embedded without introducing detectable statistical deviations.

The steganographic capacity *SC* of a stego-system is a function of the cover-object *cover* and the embedding function *emb*. SC(cover, emb) then is the maximum number of bits that can be securely embedded into *cover* by embedding function *emb* such that no detectable statistical deviations are introduced into the stego-object. In this context, securely means that the advantage of any attacker when attempting to determine the steganographic decision problem remains negligible.

Surprisingly, there has been very little work published pertaining to steganographic capacity estimates for real world steganographic systems. Notable exceptions include

the work of (Chandramouli and Memon 2001), which provide a rigorous theoretical analysis of the capacity to introduce binary distortions in the spatial domain, and the work of (Fridrich, Goljan and Hogea 2003) that briefly touches on the capacity of two well-known steganographic embedding algorithms for use on JPEG images. These analyses only provide approximations for the SC of the respective schemes, however. Determining the SC exactly for even the simplest steganographic systems remains a very difficult and as of yet unsolved problem, except for the special case when SC has previously been determined to approach 0. An example of such a case is the embedding of a steganographic message into a palette image (GIF). Such images store a reduced 128 color palette within the image. Any additionally necessary colors are then added to the palette by adding their distances from the main color palette. In doing so, the embedding of messages by making slight modifications to a pixel's color would result in clusters of close colors that are practically never created by image encoding algorithms themselves. As such, a GIF image may be determined to have been steganographically modified by simply observing the structure of its internal color palette without ever even taking into considering the image data itself.

CHAPTER 3 IMAGE STEGANOGRAPHY

3.1 Image Overview

To a computer, an image file is an array of numbers that represent light intensities at various points (pixels) (Johnson and Jajodia 1998). Color variations for the pixels are derived from the three primary colors: red, green, and blue. In a 24-bit image, the color information for each pixel is represented as three bytes, where each byte represents the intensity of one of the primary colors. In hexadecimal representation, a white pixel would be stored as the triplet (FF,FF,FF): 100 percent red intensity (FF), 100 percent green intensity (FF), and 100 percent green intensity (FF). Likewise, a black pixel would be stored as (00,00,00) and a purely red pixel would be (FF,00,00).

Figure 4: Color palette illustrating encoding of various colors using 24 bits.

Storing an image purely as a set of triplets results in a RAW image. As there is no compression of the image data, the size of the file grows linearly with the number of pixels present in the file. The need to decrease file sizes has lead to the creation of numerous image file formats. Of these, the majority seek to compress the size of the image file without noticeably degrading the quality of the image.

The various image file formats employ two types of compression: lossless and lossy. Lossless compression reduces the size of the image file while still allowing for the exact recovery of the original raw image if necessary. Examples of lossless image file formats include GIF, PNG, and BMP. Formats which employ lossy compression, on the other hand, use algorithms during the compression stage that can result in very close approximations to the original RAW image. Once a RAW image is converted into a format employing lossy compression, however, it is impossible to revert the image back to an exact copy of the original. JPEG is the most common lossy format.

3.2 Steganographic Categories

Digital image files serve as perhaps the most common cover-object for the passing of steganographic messages in the age of the personal computer. Given the plethora of image file formats, it would be impossible to comprehensively enumerate every method of hiding steganographic messages in image files. It is possible, however, to combine many unique stego-system into broad categories.

These categories include (Kharrazi, Sencar and Memon 2004):

- Spatial Domain Embedding
- Transform Domain Embedding

3.3 Spatial Domain Embedding

The steganographic principle of undetectability mandates that an embedded message must not modify its cover-object in such a way that is observable to an attacker. The result of this concept in terms of digital steganography is that the message must be embedded into the redundant bits of the cover-object. In other words, only bits whose state being flipped would cause a negligible increase in advantage for an attacker attempting to determine the steganographic decision problem may be modified during the embedding process.

In a digital image stored in a lossless format, the color intensity information for each pixel is stored as a triplet of bytes. Different values for these bytes allow the computer to differentiate between various colors. In order for the human eye to notice the difference between the two different colors, the intensity values for those colors must be significantly different. In other words, a pixel with red intensity 254 is not discernable from a pixel with red intensity 255.

Because only high order modifications result in a change to the visual appearance of an image, the least significant bits of the pixels' color intensity values can be changed without introducing visual degradation of quality. To illustrate this, an image has been modified by setting the least significant bit of every color intensity value in the entire image to 0.

Figure 5: Original Image and Modified Image with Every Color Intensity Value's LSB Set to 0

The original dimensions of the image (prior to being shrunk to fit into this document) are 480 x 720 pixels, thus there are 345,600 total pixels. Each pixel has three color intensity values R, G, and B where each color's least significant bit is flippable without visual disturbance. In such an image and only taking into account the least significant bit of each color intensity, we find there are 1,036,800 redundant bits; roughly 127 kilobytes that can be modified as desired.

While setting every color intensity's least significant bit to 0 does not visually affect the image, doing so drastically changes the statistical distribution of those bits. For example, if the least significant bit of any byte is 0 then the byte is known to be even. In setting the least significant bit of every color intensity to 0, every color intensity of every pixel was forced to be even. The probability that a digital camera would encode an image with only even intensity values is infinitesimally small, especially as the

dimensions of the image increase. Such an obvious outlying statistic would lead an observer to believe the image had been modified following its original encoding.

The previous example only took into account the least significant bit of a color intensity value. The least significant bit, however, is not necessarily the only redundancy. Figure 6 shows a bit plane decomposition of an image.

Figure 6: Bit Plane Decomposition

The top left is the original image and each following image reading to the right and wrapping around at the new line shows what happens when each subsequent least significant bit is set to 0. To clarify, the first two images share the same relationship as those in Figure 5; the original versus a modified version with every color's least significant bit set to 0. The third image is the original modified where each color's two least significant bits are set to 0. The pattern progresses sequentially until the final image in the lower right which leaves only the most significant bits of the color intensity

values intact, while setting all other bits to 0. Such an example serves to illustrate how very much an image's color information may actually be modified without drastically altering the visual image.

3.3 Transform Domain Embedding

Rather than dealing with the redundancy located in the actual color information as the spatial domain embedding methods do, transform domain embedding deals with the redundancy introduced by the various image compression algorithms themselves. In section 2.1, it was explained that JPEG is a lossy image format; that is, the encoding algorithm does not store an exact representation of the color information for each pixel, but rather it attempts to generate an accurate approximation of those colors. While negating the ability to store embedded messages directly into the color information, the application of these algorithms result in a new form of pixel descriptors. These descriptors, much like the color intensity values described in the previous section, may also be resistant to causing visual degradation when only their least significant bits are modified.

A complete explanation of the JPEG image encoding algorithm is beyond the scope of this document; however, a brief explanation of its introduction of redundancy is in order. The RGB values of the raw image's pixels are converted via an invertible *color space transform* function to a new triplet vector, whose components represent luminance, Y, and blue and red chrominance, C_b and C_r. Once converted, the algorithm

breaks the original bitmap image into 8x8 blocks of pixels. Each 8x8 block of pixels is represented by three 8x8 tables of integers containing the luminance and both sets of chrominance information. The values in these tables undergo a *discrete cosine transform*, or DCT. A DCT converts the image information to a set of average values and how much each pixel differs from this average value. Because the function is only operating on small 8x8 blocks of pixels, it is safe to assume that in many cases, the differences from the average would be relatively small and hence safely ignored (Austin 2010). The application of a DCT in conjunction with a quantization method, basically a lossy rounding function, results in 8x8 tables of DCT coefficients where the value in the upper left corner essentially represents the average over the block and values below or to the right represent horizontal and vertical frequency variations respectively.

20	-7	-1	1	-2	1	0	0
1	0	0	0	1	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 7: Possible DCT Coefficient Table after Quantization (Austin 2010)

Such a table contains many 0's, otherwise thought of as a significant lack of deviation in the pixel from the average, where what is significant is determined in the quantization step by a quality factor specified by the user. Because so many 0's are present, rather

than recording them all individually, the compression functions simply stores the number of consecutive 0's via a process called Huffman Encoding.

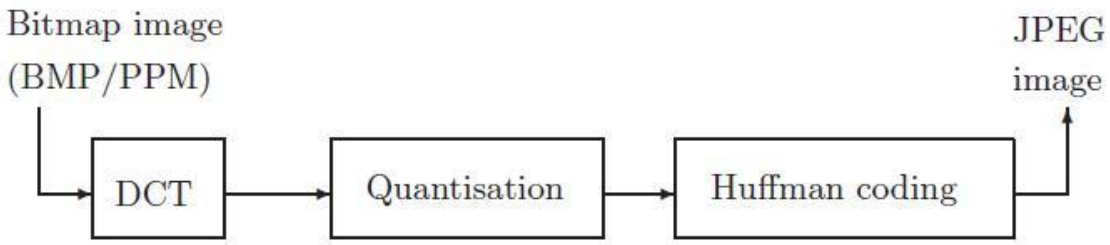


Figure 8: JPEG Encoding Process (Westfeld and Pfitzmann 2001)

It is mostly the non-zero values in the DCT coefficient tables which provide a possible location to embed steganographic messages. Small variations in these numbers, such as modifying their least significant bits, result in no more visual degradation than modifying the least significant bits of the color intensities in the spatial domain embedding examples. The reason for being unable to change the 0's themselves to hold message data is that doing so often would noticeably effect the compression rate (Kharrazi, Sencar and Memon 2004).

Numerous commercial and academic steganographic systems designed to hide messages in the DCT coefficient tables of JPEG images are available, all of which provide varying degrees of undetectability. Two such algorithms are F5 (Westfeld and Pfitzmann 2001) and Outguess (Provos 2001).

The F5 algorithm embeds messages into the least significant bits of the DCT coefficient tables, but the DCT coefficients are permuted prior to embedding. The permutation serves to spread the changed coefficients evenly over the entire image, rather than simply modifying the first *m* coefficients available. Without such a strategy, an attacker may through statistical analyses be more easily able to determine the existence of a steganographic message as the hidden information is not distributed uniformly over the image, but rather the entire change is located in one specific place.

The Outguess algorithm takes a more active approach to selecting the DCT coefficients which are appropriate for modification. The algorithm attempts to identify the redundant bits which have minimal effect on the cover-image, rather than just evenly distributing the changes throughout the file. Additionally, the authors of Outguess realized that a common steganalysis technique was to observe the distribution of DCT coefficients as a histogram. In such a scenario, if too many changes were made during the embedding process, noticeable outliers would begin to emerge. To counteract that threat, in addition to inserting the message, additional bits beyond the embedding of the actual message are also modified to readjust the stego-object's histogram back towards the cover-object's original shape.

CHAPTER 4 FILE FORMAT STEGANOGRAPHY

4.1 Introduction

For provably secure steganography to take place, the specifics of the steganographic system in use must be assumed to be known to all attackers. However in reality, there may often be occasions where such an assumption is overkill. Consider for example a situation in which the sender of a steganographic message knows ahead of time how the attacker functions. In terms of digital steganography, if a sender knows that the attacker's only method for determining if a stego-object has embedded information hidden within it is by checking whether or not the file functions as expected, then the previous considerations for security are no longer necessary.

In such a situation, it is logical for a sender to employ the simplest steganographic system possible that does not impede the original functionality of the file. To that extent, several common file formats allow for the embedding of hidden data into the file structure itself.

4.2 JPEG File Format Steganography

JPEG images are one of the most commonly found file types in modern computing. Websites generally contain numerous images and a high percentage of those images are often encoded using the JPEG format.

All JPEG images share a similar structure as defined by the JPEG File Interchange Format (JFIF) and thus contain a standard metadata header at the beginning of the file known as the JFIF segment. Following the JFIF segment header is the actual image data. The format of a JFIF segment is illustrated in Figure 9.

Field	Size (bytes)	Description
Magic Number (APP0 marker)	2	Always equals 0xFFE0
Length	2	Length of segment excluding APP0 marker
Identifier	5	Always equals "JFIF" (with zero following) (0x4A46494600)
Version	2	First byte is major version (currently 0x01), Second byte is minor version (currently 0x02)
Density Units	1	Units for pixel density fields • 00 – No units, aspect ratio only specified • 01 – Pixels per inch • 02 – Pixels per centimeter
X density	2	Integer horizontal pixel density
Y density	2	Integer vertical pixel density
Thumbnail width (tw)	1	Horizontal size of embedded JFIF thumbnail in pixels
Thumbnail height (th)	1	Vertical size of embedded JFIF thumbnail in pixels
Thumbnail data	3*tw*th	Uncompressed 24 bit RGB raster thumbnail

Figure 9: JFIF Segment Format (Wikipedia contributors, 2010)

In terms of steganography, the *Density Units*, *X density* and *Y density* are the notable fields in the JFIF section header. In order to decode and display a JPEG image, many image libraries (such as those used by Microsoft Paint, Windows Photo Gallery, Internet Explorer, and Mozilla Firefox) first parse the JFIF section header and from the Density Units, X density and Y density are able to determine in bytes the size of the image data

that follows the header. Once the size of the image data is determined, the JPEG decoding implementation reads from the file beyond the fixed size header that number of bytes and renders the pixels on the screen accordingly. Because these implementations read a pre-specified number of bytes following the header, rather than reading to End-Of-File, additional data can be appended to the end of a JPEG image that will be altogether ignored by the programs displaying the image. In this way, any JPEG image encountered could contain a hidden message at the end of the file that would remain unnoticed unless the user inspected the contents of the file with a hex editor.

It should be noted that similar ideas hold for various other common image file formats such as PNG, GIF, ICO, and certain BMP images.

4.3 ZIP File Format Steganography

The ZIP file format is a data compression and archive format and can be thought of as a container that holds one or more files that have been compressed to reduce size, or stored as is. Because the ZIP file format is an open standard as described in (PKWARE, Inc. 2007), many different implementations to handle ZIP files have been created, and of those implementations several allow for the hiding of various amounts of steganographic data within the ZIP file without an adverse effect upon its operation.

A ZIP file consists of any number of File Entries all of which share a common format, followed by a structured section known as the Central Directory. The Central Directory

contains a list of the File Entries stored in the ZIP archive along with their corresponding locations in the file (specified as relative offsets to the Central Directory). It is from this Central Directory listing that libraries implementing ZIP functionality determine the contents of the ZIP without having to parse the entire file, and then only upon the request of the user is the location of the file to be decompressed determined and subsequently extracted from the archive. A visual representation of the ZIP file format is depicted in Figure 10.

Figure 10: The ZIP File Format (Aeschbacher 2009)

Because of the addressable nature of File Entries within ZIP files as provided by the Central Directory and the fact that their location is specified via relative offset within a structure located at the end of the file, any amount of data may be prepended (appended to the beginning) to the ZIP file without invalidating the offsets stored in the Central Directory's index. As such, several ZIP implementations such as WinZIP prior to

version 8.0 and all versions of WinRAR will operate on ZIP files where data has been prepended to the file. More recent versions of WinZIP and the ZIP implementation built into Microsoft Windows no longer allow for this channel of steganographic information however, as they check (the author assumes) to ensure that the first bytes of the ZIP file are that of a valid File Entry. In order to verify this assumption, one would have to reverse engineer either WinZIP or the Windows utility to determine the algorithm, and neither program's license allows end-users to reverse engineer the software.

Beyond the steganographic channel previously described, the ZIP file format allows for another method of hiding information within it. The ZIP file format provides for the author to include a comment, which is generally an explanatory textual message to be embedded within the ZIP file. The comment is stored in the Central Directory structure, specifically at the end of it, and is allowed to be of arbitrary size. As such, the comment field is in fact located at the end of the file. Because the comment field is of variable size amongst different ZIP files, the beginning of the Central Directory structure is not located at a fixed offset from the end of the file in all ZIP files but rather is determined by the size of the comment. In order to determine the location of the Central Directory, all ZIP implementations must parse the file (most logically in reverse) until they locate the bytes that specify the beginning of the Central Directory as laid forth in the ZIP file format. Finally, there is a field in the Central Directory which specifies the size in bytes of the comment. The ZIP implementation reads the specified number of bytes at the location of the comment field and this becomes the comment as displayed to the user.

No modern ZIP implementations the author has found, however, check for data following the comment field. Similar to the JPEG image, data can again be appended to the end of the file that is altogether ignored by the programs working on the file.

Using the two steganographic channels of the ZIP file format, someone can prepend and append any amount of data to a ZIP file and it will still function properly with WinZIP prior to version 8.0 and all versions of WinRAR.

4.4 Combining File Format Steganography

In the previous two sections, it has been shown that any amount of data may be appended to the end of a JPEG image and likewise data may be prepended and appended to a ZIP file, where doing so does nothing to impede the expected functionality of said files. Such a scenario gives rise to a steganographical experiment: Is it possible, using the steganographic idiosyncrasies of the previously described file formats, to combine files of those formats in such a way that both remain functional at the same time? In fact, it is absolutely possible to do so.

Because libraries that work with JPEG images only read a certain number of bytes from the beginning of the file as specified by the header and ZIP files determine the location of File Entries from a relative location as specified towards the end of the file, a ZIP file may be appended to the end of a JPEG image. Subsequently, if the resulting file is

opened within an image viewer, the JPEG image will be displayed. Alternatively, if the file is opened in a program that manipulates ZIP files, it will also work as expected.

An interesting example of such a system is included in the Associated Files of this document. A ZIP file called source.zip containing an unmodified image, source.jpg, was generated. A new file called merged.jpg was created by appending the binary contents of source.zip past the end of source.jpg. The file merged.jpg opens in an image editor and displays the exact same image is source.jpg. However, if merged.jpg is renamed to merged.zip, it may be opened in WinZIP prior to version 7.0 or WinRAR and the original source.jpg may be extracted back out. In summary, a single file was generated that may be viewed as an image or may, via the application of the ZIP extraction algorithm, extract the same image out of it.

CHAPTER 5 COMMONSTEG

5.1 Introduction to CommonSteg

CommonSteg is a computer program written in Visual C# in order to illustrate several practical steganographic techniques outlined in this document. Pertaining to images, it contains algorithms to embed one image into the least significant bits of another, embed text into the least significant bits of an image either sequentially or via a keyed "random walk," and also has the ability to extract the messages embedded via those algorithms back out of the stego-objects. CommonSteg also provides the ability to easily append data to the beginning or end of any file, thereby providing support for the techniques listed in the File Format Steganography section.

The user's manual for CommonSteg can be found in Appendix B, and the source code in Appendix C.

5.2 Hiding Images within Images

CommonSteg contains functionality to embed a hidden image into a secondary cover-image. The algorithm to embed one image into the least significant bits of another is defined as follows:

Image Embedding Algorithm

(1) Check that both the cover-image and message image have equal dimensions. If so, continue to (2), else exit.
(2) For each pixel in the message image, perform the following steps:
    a. Determine the intensity of the redness of the pixel via its RGB values. Intensity is said to be high if 255 >= R >= 192, mid if 191 >= R >= 128, low if 127 >= R >= 64, and zero if 64 > R.

b. Determine the intensity of the greenness of the pixel using the same methodology as for redness.
c. Determine the intensity of the blueness of the pixel using the same methodology as for redness.
d. At the same pixel location in the cover-image, for each of the three colors:
    i. If the color intensity in the message image is high, then replace the 2 least significant bits of the color's byte with 11.
    ii. If the color intensity in the message image is mid, then replace the 2 least significant bits of the color's byte with 10.
    iii. If the color intensity in the message image is low, then replace the 2 least significant bits of the color's byte with 01.
    iv. If the color intensity in the message image is zero, then replace the 2 least significant bits of the color's byte with 00.

More generally, the embedding process works by categorizing the intensity of each of the three primary colors into four distinct ranges. The range into which the color falls is then embedded into the least significant bits of the corresponding pixels in the cover-image.

In performing this algorithm, the visual quality as perceived by the human eye of the subsequently produced stego-image remains unchanged. The quality of the embedded message, however, is likely significantly degraded via the process. Because the intensity values for each pixel are encoded using only two bits for each color, the process reduces the palette of the message image from whatever its original color count was to a total of 64 possible colors; that is, there are four possible shades for each of the three primary colors which can be combined in 4 x 4 x 4 = 64 possible ways.

The retrieval algorithm functions in the reverse manner. The least significant bits for each color in each pixel of the stego-image are read and subsequently amplified. For example, if during the embedding algorithm a pixel was determined to have a high red intensity (red intensity >= 192), then the two least significant bits in the byte were set to 11. These bits are then amplified by bit-shifting the byte 6 positions to the left, resulting in 11000000 (192 decimal). Similarly, a pixel with color intensity in the low range would have had its least significant bits set to 01, which upon amplification would result in a color intensity of 01000000 (64 decimal). If all of the more significant bits were simply discarded, rather than implementing the amplification process, the resulting image would only contain color intensities of 0, 1, 2, and 3 (all decimal) respectively. Such a result would provide no distinction between the colors in terms of what is visible to the human eye.

Image Retrieval Algorithm

(1) For each pixel in the stego-image, perform the following steps:
    a. For each color intensity value Red, Green, Blue:
        i. Obtain the intensity of the color (as a byte).
        ii. Perform a 6-digit bitwise left shift of the byte.
        iii. Set the resulting value as the new color intensity.

The following two images served as inputs to CommonStegs image embedding algorithm, representing the cover-image and the message image to be embedded into the cover respectively.

Figure 11: Cover-Image and Message to be Embedded

The algorithm output the following visually identically stego-image. The image retrieval algorithm subsequently recreated a 64-color approximation of the original message.

Figure 12: Stego-Image and Retrieved Message

The retrieved message, while being noticeably degraded in quality, is still perfectly identifiable.

5.3 Hiding Text within Images

In addition to hiding an image within another, CommonSteg also has the ability to hide ASCII text messages within the least significant bits of a cover-image where not only is

there no visual degradation of quality of the outputted stego-image, but the extracted message itself remain fully intact.

The algorithm works by obtaining the ASCII character code for each character in the text, converting it to a byte, and starting from the most significant bit encodes the value sequentially into the least significant bit of the red intensity value of the image's pixels, starting from the upper left and wrapping at the end of a row. Finally, eight 1's are embedded immediately beyond the end of the actual message in order to represent the end of the message. This results in a string of characters terminated by a character with ASCII value 255. The value 255 was chosen arbitrarily to serve as the terminator, and may not be ideal in terms of avoiding deviations from an observable distribution of least significant bits. In theory, any value outside the range of printable ASCII characters would work just as well as 255.

Embed Algorithm

(1) User selects cover image and message text. For the total pixel count *t* of the cover image and the message length *n*, if t >= (n*8)+8 is true, continue to (2), else exit.
(2) For each character in the message text, perform the following:
    a. Obtain the ASCII code of the character (a byte) and convert it to binary.
    b. For each bit in the byte, starting from the most significant, perform the following:
        i. Starting from the most upper-left pixel of the cover image reading to the right and wrapping around at the end of each row, replace the least significant bit of the next unchanged pixel's red intensity value with the message bit.
    c. Starting at the pixel immediately following the final pixel modified in step b, set the least significant bit of the red intensity for eight consecutive pixels to 1.

Retrieval Algorithm

(1) While the character *c* retrieved from the stego-image does not have ASCII value 255, perform the following steps:
    a. Let c = 0 and *ii* = 7. While ii >= 0, do:
        i. Starting from the most upper-left pixel of the cover image reading to the right and wrapping around at the end of each row, obtain the red intensity value of the next previously unread pixel.
        ii. If the red intensity value is odd, set c = c + 2^ii.
        iii. Decrement ii by 1.
    b. If c is not 255, convert the byte via the corresponding ASCII code to a character and concatenate it to the extracted message string.
(2) Output extracted message string to the user.

While the algorithm described does generate a visually equivalent stego-image, if an attacker is aware of the algorithm used to embed the message, determining whether or not an embedded message is present is trivial.

5.4 Hiding Text in Images More Securely with Random Walk

By Kerckhoffs' Principle, a steganographic scheme should be secure even if its algorithms are known by the attacker; that is, the security should rely entirely on the stego-key. In order to rectify this problem, a second technique for steganographically embedding text into an image is provided in CommonSteg. This second set of algorithms is more secure because rather than embedding the message bits consecutively and in order starting at the upper-left of the image, the pixels containing the message bits are distributed pseudo-randomly throughout the image.

To accomplish this, a pseudorandom number generator (PRNG) is employed to determine the specific pixels that are to hold the message bits. A PRNG is an algorithm

which attempts to produce a sequence of independent random numbers with a specified distribution (Knuth 1969). The PRNG takes as input an arbitrary seed value that sets the initial state of the algorithm and outputs a sequence of random integers within the bounds as specified by the user. Providing a PRNG with the same seed will result in the same sequence of pseudorandom numbers every time. In this case, the default PRNG provided by Microsoft in the .NET Framework Random Class Library, Donald Knuth's Subtractive Random Number Generator algorithm (Microsoft Corporation 2010), is used in conjunction with a user provided seed value to generate a sequence of integers from a uniform distribution within the range of zero and the total count of pixels in the cover image minus one inclusive. The numbers in this sequence are then converted to (X,Y) pixel coordinates which subsequently determine the locations to insert the message bits.

Because a PRNG always generates the same sequence when it is initialized with the same seed value, the recipient of the stego-object is able to ascertain the locations the message bits if he knows said seed value. As such, in this steganographic scheme, the seed value acts as a stego-key. Because any pixel in the image is equally as likely as any other to have been selected to hold a message bit and also because determining the locations and ordering of those message-holding pixels now requires a shared secret between the sender and the recipient, ascertaining whether or not a possible stego-object contains an embedded message is no longer trivial for an attacker, even if he is aware of the algorithm via which it was embedded.

Rather than following a predefined location pattern as in the first text hiding algorithm, a steganographic scheme is said to employ a "random walk" if the locations into which the message bits are hidden are determined with a degree of randomness.

Random Walk Embed Algorithm

(1) User selects cover image, message text, and a seed value *s*. If the seed value is not a valid 32-bit signed integer, exit. For the total pixel count *t* of the cover image and the message length *n*, if t >= (n*8)+8 is true, continue to (2), else exit.
(2) For each character in the message text, perform the following:
    a. Obtain the ASCII code of the character (a byte) and convert it to binary. For each bit in the byte, do:
        i. Using the PRNG initialized with *s*, generate the next random number *k* in the sequence within the bounds of zero and the total pixel count of the image minus one inclusive. If *k* has already been seen in the sequence, discard it and repeat step b.
        ii. Where *w* is the width of the cover image in pixels, convert *k* to an (x,y) coordinate as follows:
            1. x = k % w where % is the modulus operation.
            2. y = k / w where / is the division operation that only returns whole numbers.
        iii. For the pixel at coordinate (x,y), replace the least significant bit of the red intensity value with the message bit.
(3) Perform the following eight times:
    a. Continuing to use the PRNG initialized with *s*, generate the next random number *k* in the sequence within the bounds of zero and the total pixel count of the image minus one inclusive. If *k* has already been seen in the sequence, discard it and repeat step b.
    b. Convert *k* to (x,y) coordinates using the same process as before.
    c. For the pixel at coordinate (x,y), replace the least significant bit of the red intensity value with 1.

Random Walk Retrieval Algorithm

(1) Input PRNG seed value *s* from the user.
(2) While the byte *c* retrieved from the stego-image is not 255, perform the following steps:

a. Let c = 0 and *ii* = 7. While ii >= 0, do:
    i. Using the PRNG initialized with *s*, generate the next random number *k* in the sequence within the bounds of zero and the total pixel count of the image minus one inclusive. If *k* has already been seen in the sequence, discard it and repeat step i.
    ii. Convert *k* to an (x,y) coordinate via the same process as in the embedding algorithm.
    iii. Read the red intensity value of the pixel at (x,y). If it is odd, set c = c + 2^ii.
    iv. Decrement ii by 1.
b. If c is not 255, convert the byte via the corresponding ASCII code to a character and concatenate it to the extracted message string.

(3) Output extracted message string to the user.

5.5 Security Analysis of Random Walk Algorithm

It is obvious that many of steganographic techniques demonstrated herein have little to zero steganographic security in terms of the security frameworks discussed previously. This is because if the attacker knows the steganographic system used to embed the message, it is trivial for him to ascertain whether or not such a message is present in any given stego-object. That is not the case for CommonSteg's Random Walk Textual Embedding steganographic scheme, however, due to the introduction of a stego-key.

In terms of Zöllner's information theoretic security framework, the Random Walk stego-system is far from perfectly secure. For one, the stego-key must be drawn from a bounded domain: the set of signed 32-bit integers greater than or equal to 0. This range, 0 to +2,147,483,647, is well within the brute-forcible capacity of a modern PC's computational power. In addition, if the attacker knows with relative accuracy the distribution of the least significant bits of the red intensities of the pixels in images from

the source, which in this case would be the set of images encoded into the PNG format, statistical analyses on the pixels of possible stego-images might reveal deviations from the norm. Such deviations would lead the attacker to believe that the image might contain a steganographic message. Because the amount of deviation from the standard distribution is directly related to both the length of the embedded message and the dimensions of the image into which it is to be embedded, special consideration must occur when selecting a cover-object to hold a message of a particular length. A steganographer who embeds a single character message into an image using CommonSteg's embedding algorithm will modify at most 16 bits within the image; 8 bits for the character and an additional 8 for the terminator. Exactly 16 bits need not always be modified as there is a chance that a pixel chosen by the PRNG to hold the message bit already matches it. Given a sufficiently large image, it is safe to say that the deviation from the source distribution introduced by modifying such a small number of bits would be within the error range of the attacker's source distribution.

Another consideration for practical security was that an attacker should be able to differentiate between actual messages embedded via the steganographic scheme and other artifacts which merely appear to suggest the existence of a message. While a combination of brute-forcing the stego-key and a heuristic technique for determining whether or not a string of characters is plain English might very well determine the existence of a sentence embedded into a stego-object, such an attack could not succeed with any significant degree of accuracy if the length of the message contained was

limited to a relatively short string of characters. In other words, as long as the embedded message is sufficiently short, such as in the case of a single previously agreed upon character, a heuristic algorithm for determining whether or not a decoded message is English would inherently provide a negligible advantage in successfully deciding the steganographic decision problem. Even if an attacker were to brute-force every possible stego-key, the likelihood of finding numerous single valid ASCII characters is so high that identifying any of them as an actual steganographic message would be an unreasonable assumption. As such, I therefore conclude that for sufficiently short messages relative to the dimensional size of the cover-image, CommonSteg's Random Walk embedding algorithm is conditionally secure.

To support this conclusion, a brute-forcer designed to attack the Random Walk algorithm is also included in CommonSteg. The attack functions by retrieving the message embedded for each possible stego-key. If a returned message is found to only contain valid ASCII characters, then it is logged. However, if a character outside of the range of valid ASCII characters is found, that seed is immediately invalidated and the attack code proceeds to the next. To conduct the experiment, a cover-image with no steganographically hidden message was selected. The attack retrieved and counted all valid ASCII messages for stego-keys in the range of 0 through 100,000,000 inclusive. See Appendix D for the detailed results.

In summary, the attack conducted on a standard image with dimensions 604 x 453 (pixels) found that 234,875 out of the 100,000,001 attempted stego-keys resulted with the return of an embedded message containing only valid ASCII characters. The somewhat optimized attack took 1752 seconds to complete on a 2.66 GHz Intel Core 2 Quad CPU with 4 GB of RAM available. This rate can be extrapolated to determine the duration of the time needed to brute-force the entire keyspace on such a machine: roughly ten and a half hours. Since no steganographic message was actually hidden within the image, all 234,875 ASCII messages that were logged were false positives. This gives an estimated false positive occurance rate of 234,875 / 100,000,001 ≈ 0.23%. Such a rate implies that the image would roughly contain 2,147,483,648 * 0.23% ≈ 4,939,212 false positives.

Again, Appendix D shows a sample listing of the false positives found within the image. It is obvious that a grammatically structured plaintext message would stand out distinctly from the list of false positives present, the majority of which are no longer than a single character or two and make no sense in terms of formal language. However, if the message could be conveyed in only a single character (perhaps y or n to indicate yes or no), then access to such a log of all valid ASCII character messages contained within the stego-object should provide negligible advantage when determining the steganographic decision problem.

CHAPTER 6 Conclusions

6.1 Usage Estimates

In 2001, USA Today reported that terrorists organizations such as Al Qaeda were using steganography to covertly deliver instructions and messages to their followers (USA Today 2001). Specifically, it was stated that U.S. officials confirm that terrorist messages had been steganographically embedded into X-rated pictures on several pornographic websites. In response, researchers at the University of Michigan attempted to determine the presence of steganographic activity on the internet by scraping JPEG images from multiple popular sources and subjecting those images to steganalysis (Provos and Honeyman 2001). Two million JPEG images were scraped from eBay.com and an additional one million were culled from Usenet. All of the images were subjected to a series of tests designed to identify steganographic embedding by four common steganography programs. For those images that tested positive for suspicion of containing an embedded message, a comprehensive dictionary attack was employed to determine if a message was actually present within the image. To this date, and after scanning over three million total images, they have not found a single hidden message.

From this statistic, one can infer that obtaining steganography usage estimates is an inherently difficult task. Because the entire purpose of steganography is the undetectability of its usage and as those steganographic techniques become more advanced, it becomes an increasingly impossible task to accurately quantify the amount of usage actually taking place. While accurately quantifying usage may be outside of the

realm of possibility, it is reasonable to assume that steganographic communications to some degree are taking place simply by the sheer prevalence of information related to steganographic techniques. A Google search for the term Steganography returns about 447,000 pages containing the term at the time of writing and a search of scholarly publications via scholar.google.com with the key word steganography results in roughly 16,500 articles.

6.2 Additional Applications of Steganographic Techniques

Steganography was defined as the art and science of writing hidden messages in such a way that no one, apart from the sender and intended recipient, suspects the existence of the message. There are, however, many other scenarios where the usage of steganographic techniques such as those described herein are applicable, yet do not fit perfectly into that definition. For example, there is no need for that which is hidden to be a message (rather than data) and also it might very well be the case that the creator of a file containing steganographic information may not want the recipient, if there even is one, to know of the hidden data's presence. These realizations give rise to two related yet slightly different studies: data hiding and watermarking.

In a data hiding scenario, a user may attempt to hide some private information from all other parties by embedding it steganographically into some innocuous file. While making use of a steganographic embedding function to perform that task, the act of

doing so does not actually fall under the standard definition of steganography because there is no other intended recipient and as such the corresponding stego-object is never sent anywhere.

The inverse of the data hiding scenario is that of the watermarking scenario. In this case, a content provider may wish to identify the source of a file if it is leaked beyond its intended recipients. To do so, the provider may make use of a steganographic embedding function to hide an identifying marker such as the recipient's name into the file. If the file is subsequently leaked, the provider may retrieve the watermark from the file in order to determine who was responsible. Such watermarking techniques are often employed for the purposes of intellectual property control. A prime example is the watermarking of Hollywood movies, whereby the producers embed recognizable artifacts into the movies which may be used to identify the theater to which it was sent. In doing so, the source of movies pirated onto the internet may be determined. The study of creating robust watermarks resistant against an adversary attempting to remove them is currently an active field.

Due to their significant overlap, the three distinct fields of steganography, data hiding, and watermarking might best be combined into one related topic called covert information hiding. A definition of such a new field might be: The art and science of hiding information in such a way that no one apart from the creator and those he informs suspects the existence of the information. Such a new definition removes the

unnecessary qualifiers that lead to the alternate descriptions of such closely related topics in the first place.

6.3 Final Thoughts

The evolving practice of steganography has been in use for thousands of years. From its humble beginnings on a man's scalp, the techniques available to covertly deliver messages continue to increase in sophistication as the desire for greater security against being found out rises. As such, steganography and its related branches of data hiding and watermarking are increasingly active fields in the disciplines of Computer Science and Information Theoretics.

Several other academic entries relating to steganography have ended with a reference Poe's "The Purloined Letter", implying that sometimes hiding your message in plain sight might be the better option. Inclined to do the same, I reference the story itself:

> "*The more I reflected upon the daring, dashing, and discriminating ingenuity of D-- . . . the more satisfied I became that, to conceal this letter, the Minister had resorted to the comprehensive and sagacious expedient of not attempting to conceal it at all.*"

The parallel to steganography is obvious, notwithstanding Poe's misstatement: that D-- made no effort to conceal the letter, when in fact he wrote a new address on the back of the stolen one, refolded it the opposite way, and sealed it with his own seal. Hiding a message within an innocuous cover object is really no different than resealing a letter

with your own stamp and leaving it in the open. It is just important not to forget that an amateur detective saw right through D--‘s ploy.

Bibliography

Aeschbacher, N. *The ZIP64File Java Library, Technical Documentation, Version 1.2.* 2009. http://www.enterag.ch/enterag/downloads/Zip64File_TechnicalDocumentation.pdf.

Austin, D. *Image Compression: Seeing What's Not There.* 2010. http://www.ams.org/samplings/feature-column/fcarc-image-compression (accessed April 4, 2010).

Cachin, Christian. "Digital Steganography." In *Encyclopedia of Cryptography and Security*. Springer Verlag, 2005.

Chandramouli, R., and N. Memon. "Analysis of LSB Based Image Steganography Techniques." *Proc. of ICIP 2001.* Thessaloniki, Greece, 2001.

Fridrich, J., M. Goljan, and D. Hogea. "New methodology for breaking steganographic techniques for JPEGs." *SPIE Symposium on Electronic Imaging.* Santa Clara, CA, 2003.

Gallager, R.G. *Information Theory and Reliable Communication.* New York: John Wiley & Sons, 1968.

Goldwasser, S., S. Micali, and P. Tong. "Why and how to establish a private code on a public network." *Proc. 23rd IEEE Symp. on Foundations of Comp. Science.* Chicago, 1982. 134-144.

Johnson, N.F., and S. Jajodia. "Exploring Steganography: Seeing the Unseen." *IEEE Computer*, 1998: 26-34.

Judge, J.C. *Steganography: Past, Present, Future.* November 30, 2001. http://www.sans.org/rr/papers/index.php?id=552 (accessed April 14, 2010).

Katzenbeisser, S., and F.A. Petitcolas. *Information hiding techniques for steganography and digital watermarking.* Norwood, MA: Artech House Books, 1999.

Katzenbeisser, S., and F.A.P Petitcolas. "On Defining Security in Steganographic Systems." *Proceedings of SPIE, Security and Watermarking of Multimedia Contents IV.* San Jose, CA: International Society for Optical Engineering, 2002. 50-56.

Kharrazi, M., H.T. Sencar, and N. Memon. *Image Steganography: Concepts and Practice.* Lecture Note Series, Singapore: Institute for Mathematical Sciences, National University of Singapore, 2004.

Knuth, D.E. *The Art of Computer Programming, Vol 2: Seminumerical Algorithms.* Reading, Mass.: Addison-Wesley, 1969.

Madsen, Wayne, and David Banisar. *Cryptography and Liberty 2000 - an International Survey of Encryption Policy.* Washington DC: Electronic Privacy Information Center, 2000.

Microsoft Corporation. *.NET Framework Class Library - Random Class.* 2010. http://msdn.microsoft.com/en-us/library/system.random.aspx (accessed May 8, 2010).

Pfitzmann, B. "Information hiding terminology." *Information Hiding, First International Workshop, vol. 1174 of Lecture Notes in Computer Science.* Springer, 1996. 347-350.

PKWARE, Inc. *APPNOTE.TXT - .ZIP File Format Specification. Version 6.3.2.* January 28, 2007. http://www.pkware.com/documents/casestudies/APPNOTE.txt.

Princeton University. *WordNet Search.* http://wordnetweb.princeton.edu/perl/webwn?s=information%20theory (accessed May 4, 2010).

Provos, N. "Defending Against Statistical Steganalysis." *10th USENIX Security Symposium*, 2001.
Provos, N., and P. Honeyman. "Detecting Steganographic Content on the Internet." *CITI Technical Report 01-11*, 2001.
Simmons, G. J. "The prisoners' problem and the subliminal channel." *CRYPTO'83.* Plenum Press, 1984. 51-67.
Trithemius, Johannes. *Steganographia: Hoc est: Ars per occultam, etc.* 1621. http://www.esotericarchives.com/tritheim/stegano.htm (accessed March 4, 2010).
Unknown. *Image Steganography and Steganalysis.* 2007. http://www.ims.nus.edu.sg/Programs/imgsci/files/memon/sing_stego.pdf (accessed March 4, 2010).
USA Today. *Terror groups hide behind Web encryption.* February 5, 2001. http://www.usatoday.com/tech/news/2001-02-05-binladen.htm (accessed May 8, 2010).
Westfeld, A., and A. Pfitzmann. "High Capacity Despite Better Steganalysis (F5-A Steganographic Algorithm)." *Lecture Notes in Computer Science* (Springer-Verlag) 1768 (2001): 61-75.
Wikipedia contributors. *Chosen-plaintext attack.* Wikipedia, The Free Encyclopedia. February 10, 2010. http://en.wikipedia.org/w/index.php?title=Chosen-plaintext_attack (accessed April 18, 2010).
—. *Information theory.* May 10, 2010. http://en.wikipedia.org/w/index.php?title=Information_theory (accessed May 13, 2010).
—. *RSA.* April 8, 2010. http://en.wikipedia.org/w/index.php?title=RSA (accessed April 23, 2010).
Zöllner, J., et al. "Modeling the security of steganographic systems." Edited by D. Aucsmith. *Information Hiding, 2nd International Workshop.* Spring, 1998. 344-354.

APPENDIX

Appendix A – RSA Public Key Cryptography

RSA is an algorithm for public key cryptography, invented by cryptographers Rivest, Shamir, and Adleman. It works as follows (quoted directly from (Wikipedia contributors 2010)):

Key Generation

RSA involves a public key and a private key. The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key. The keys for the RSA algorithm are generated in the following way:

1. Choose two distinct prime numbers p and q.
    - For security purposes, the integers p and q should be chosen uniformly at random and should be of similar bit-length. Prime integers can be efficiently found using a primality test.
2. Compute $n = pq$.
    - n is used as the modulus for both the public and private keys
3. Compute $\varphi(pq) = (p - 1)(q - 1)$. ($\varphi$ is Euler's totient function).
4. Choose an integer e such that $1 < e < \varphi(pq)$, and e and $\varphi(pq)$ share no divisors other than 1 (i.e., e and $\varphi(pq)$ are coprime).
    - e is released as the public key exponent.
    - e having a short bit-length and small Hamming weight results in more efficient encryption. However, small values of e (such as $e = 3$) have been shown to be less secure in some settings.
5. Determine d (using modular arithmetic) which satisfies the congruence relation $de \equiv 1 \pmod{\varphi(pq)}$
    - Stated differently, $ed - 1$ can be evenly divided by the totient $(p - 1)(q - 1)$.
    - This is often computed using the extended Euclidean algorithm.
    - d is kept as the private key exponent.

The **public key** consists of the modulus n and the public (or encryption) exponent e. The **private key** consists of the private (or decryption) exponent d which must be kept secret.

Encryption

Alice transmits her public key (*n*,*e*) to Bob and keeps the private key secret. Bob then wishes to send message **M** to Alice.

He first turns **M** into an integer $0 < m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext *c* corresponding to:

$$c = m^e \bmod n$$

This can be done quickly using the method of exponentiation by squaring. Bob then transmits *c* to Alice.

Decryption

Alice can recover *m* from *c* by using her private key exponent *d* by the following computation:

$$m = c^d \bmod n.$$

Given *m*, she can recover the original message **M** by reversing the padding scheme.

(In practice, there are more efficient methods of calculating c^d using the pre computed values above.)

Appendix B – Brute Force Attack Output For CommonSteg's Random Walk Algorithm

The following is a trimmed output of CommonSteg's brute force attack against the image described in 3.4. While the attack logged all messages containing only valid ASCII characters for stego-keys between the range of 0 and 100,000,000, for brevity the following results only show the corresponding messages for stego-keys < 100,000.

234875 Messages With Valid ASCII Codes Were Found in 1752 Seconds
Seed Range: 0 to 100000000

Seed: 197 Message: v
Seed: 1167 Message: e
Seed: 1187 Message: *
Seed: 1813 Message: >
Seed: 1972 Message: S
Seed: 2223 Message: w
Seed: 2253 Message: ?|
Seed: 3154 Message: %
Seed: 3262 Message:)w$
Seed: 3806 Message: qiteC
Seed: 4133 Message: H
Seed: 4446 Message: S
Seed: 4615 Message: z
Seed: 4676 Message: C~
Seed: 5395 Message: I.FlG
Seed: 5690 Message: ;7
Seed: 5720 Message: =
Seed: 5940 Message: guTt'
Seed: 7386 Message: v>
Seed: 8360 Message: S
Seed: 8460 Message: D
Seed: 8589 Message: u
Seed: 8651 Message: 1
Seed: 9402 Message: NH
Seed: 9585 Message: b
Seed: 11050 Message: C?
Seed: 11931 Message: `
Seed: 12435 Message:
Seed: 13128 Message: .
Seed: 13509 Message: 7
Seed: 13663 Message: \>
Seed: 13923 Message: i
Seed: 14214 Message: @
Seed: 14939 Message: R
Seed: 15782 Message: l
Seed: 16365 Message: q52
Seed: 16823 Message: _j
Seed: 17207 Message: =`
Seed: 17350 Message: &

Seed: 17529 Message: #
Seed: 17547 Message: d
Seed: 17556 Message: 6
Seed: 18410 Message: `
Seed: 18618 Message: T
Seed: 19301 Message: im
Seed: 19573 Message: C
Seed: 19698 Message: D$
Seed: 20325 Message: U
Seed: 20911 Message: '
Seed: 21032 Message: F*
Seed: 21181 Message: /.N
Seed: 21627 Message: T
Seed: 21879 Message: /
Seed: 21943 Message: s
Seed: 22010 Message: {
Seed: 22131 Message: *B
Seed: 22270 Message: $
Seed: 22561 Message: L
Seed: 23054 Message: M
Seed: 23552 Message: (
Seed: 23674 Message: [v
Seed: 23830 Message: d
Seed: 24917 Message: >
Seed: 25605 Message: S
Seed: 25774 Message:)
Seed: 26160 Message: {%
Seed: 26814 Message: 0~
Seed: 27257 Message: _
Seed: 27520 Message: C#
Seed: 27577 Message: B
Seed: 27975 Message: I
Seed: 27981 Message: ,
Seed: 28814 Message: S
Seed: 28869 Message: b
Seed: 29141 Message: gB
Seed: 30794 Message: "
Seed: 30973 Message: {6
Seed: 31096 Message: X
Seed: 31303 Message: z
Seed: 31565 Message: o
Seed: 32039 Message: IP
Seed: 32522 Message: p
Seed: 33142 Message: h
Seed: 33171 Message: Q
Seed: 33474 Message: 7y
Seed: 33946 Message: ;
Seed: 34081 Message: 1
Seed: 34138 Message: WG
Seed: 34217 Message: C
Seed: 34331 Message: 8
Seed: 34418 Message: nLB&U
Seed: 35556 Message: f

Seed: 36052 Message: |
Seed: 36096 Message: [
Seed: 36490 Message: j
Seed: 36492 Message: 0"
Seed: 36817 Message: y
Seed: 36958 Message: a
Seed: 37027 Message: R
Seed: 37199 Message: AW
Seed: 37237 Message: b
Seed: 37359 Message: >
Seed: 38135 Message: [jo
Seed: 38380 Message: B
Seed: 38714 Message: V
Seed: 38908 Message: .}
Seed: 39264 Message: Z
Seed: 39925 Message: }
Seed: 39946 Message: EkY
Seed: 40252 Message: >O
Seed: 40624 Message: c
Seed: 41795 Message: B
Seed: 41803 Message: @
Seed: 41930 Message: .
Seed: 43175 Message: >\
Seed: 43974 Message: S]2
Seed: 44492 Message: G
Seed: 44654 Message: p
Seed: 44749 Message: Ze
Seed: 44771 Message: J
Seed: 45751 Message: v
Seed: 45925 Message: v
Seed: 46055 Message: ^
Seed: 46056 Message: @
Seed: 46367 Message: ac
Seed: 46714 Message: 9
Seed: 46840 Message: QV
Seed: 47653 Message: U
Seed: 47738 Message: _
Seed: 47983 Message: }
Seed: 48229 Message: VG
Seed: 48300 Message: $
Seed: 48989 Message: }7B
Seed: 49005 Message: *
Seed: 49244 Message: g
Seed: 49562 Message: y
Seed: 50052 Message: &`
Seed: 50300 Message: 7
Seed: 51660 Message: '
Seed: 51983 Message: J
Seed: 52154 Message: m
Seed: 52418 Message: A
Seed: 52776 Message: t
Seed: 53007 Message: R
Seed: 53107 Message: <

Seed: 53168 Message: D
Seed: 53295 Message: +}
Seed: 55407 Message: aD
Seed: 55700 Message: G
Seed: 55978 Message: f
Seed: 56447 Message: S
Seed: 56647 Message: CE
Seed: 56936 Message: ?
Seed: 57135 Message:]
Seed: 57234 Message: 7}?
Seed: 57474 Message: B
Seed: 58045 Message: H
Seed: 58174 Message: T
Seed: 58785 Message: =
Seed: 58870 Message: '
Seed: 58877 Message: 0
Seed: 59351 Message: ,
Seed: 59594 Message: W@v
Seed: 60320 Message: 9E
Seed: 60381 Message: B&
Seed: 61188 Message: w.6~
Seed: 61776 Message: 7
Seed: 62136 Message: P[
Seed: 62156 Message: $_
Seed: 62194 Message: S
Seed: 62790 Message: =
Seed: 63757 Message: !
Seed: 65252 Message: xN
Seed: 65410 Message: j
Seed: 65583 Message: #
Seed: 65641 Message: O<
Seed: 66596 Message: I
Seed: 67754 Message: C2
Seed: 67816 Message: B)
Seed: 67817 Message: B
Seed: 68293 Message: G
Seed: 68443 Message: Ss
Seed: 68540 Message: }F
Seed: 69189 Message: k
Seed: 69460 Message: na
Seed: 69714 Message: l
Seed: 70103 Message: -
Seed: 70428 Message: p}l
Seed: 70709 Message: IcmvU
Seed: 71355 Message:
Seed: 71879 Message: r
Seed: 73142 Message: b
Seed: 73143 Message: 8
Seed: 73168 Message: }
Seed: 73505 Message: I
Seed: 74250 Message: S
Seed: 75541 Message: ^
Seed: 75675 Message: Q

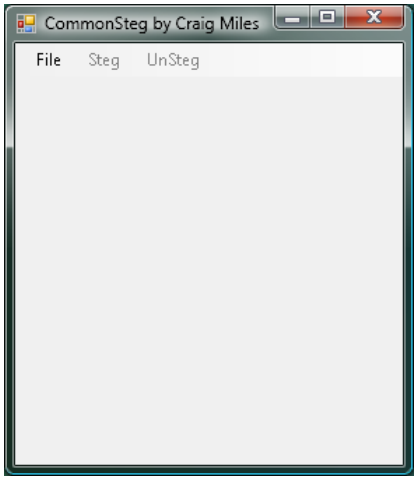
Seed: 76285 Message: I
Seed: 77395 Message: ;
Seed: 77841 Message: R
Seed: 78168 Message: _
Seed: 78213 Message: 1F;
Seed: 78368 Message: {
Seed: 78602 Message: u
Seed: 79449 Message: D
Seed: 79658 Message:]
Seed: 80039 Message: \
Seed: 80962 Message: z6
Seed: 81298 Message: (Z
Seed: 81766 Message: E
Seed: 82204 Message: >;u
Seed: 82590 Message: `y
Seed: 82954 Message:)
Seed: 83736 Message: N
Seed: 83835 Message: >
Seed: 84021 Message: ,
Seed: 84327 Message: 2)%
Seed: 84957 Message: [v
Seed: 85807 Message: F
Seed: 85990 Message: vfW
Seed: 86526 Message: h
Seed: 86652 Message: L
Seed: 87319 Message: O
Seed: 87509 Message: /
Seed: 87702 Message: qQ
Seed: 88271 Message: E
Seed: 88835 Message: h
Seed: 90002 Message: =
Seed: 90127 Message: :T
Seed: 90768 Message: *
Seed: 91411 Message: <c
Seed: 91481 Message: ^A
Seed: 92078 Message: '
Seed: 92211 Message: d
Seed: 92252 Message: H"
Seed: 92443 Message: +&(
Seed: 92526 Message: O
Seed: 93688 Message: _
Seed: 93991 Message: 5
Seed: 94610 Message: iJ
Seed: 95664 Message: !
Seed: 95727 Message: H
Seed: 95876 Message: =
Seed: 95962 Message: -
Seed: 95991 Message: m
Seed: 96018 Message: c;
Seed: 96632 Message: =d
Seed: 96910 Message: ,
Seed: 97157 Message: d
Seed: 97327 Message: >

Seed: 97525 Message: B
Seed: 97774 Message: 6|
Seed: 98034 Message: 9
Seed: 98151 Message: Q,v
Seed: 98170 Message: Z
Seed: 98709 Message: /Sa
Seed: 98861 Message: :
Seed: 99092 Message: S
Seed: 99299 Message: Nr
Seed: 99426 Message: !4
Seed: 99666 Message: 1

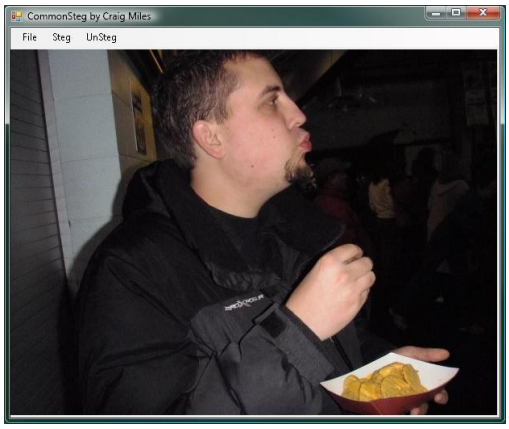
Appendix C – CommonSteg User's Manual

File Menu

Upon starting CommonSteg, you are presented with a blank image plane.



An image must be opened before any other functionality may take place. Open an image by using the File -> Open dialog. Once opened, the program window's size is automatically adjusted to accommodate the size of the image.



Now that a picture is opened, the Steg and Unsteg menu items become available. Also, the Save As functionality becomes enabled. By using the File -> Save As dialog, the image currently in the display window may be saved as a new file.

Steg Menu

The Steg menu provides three options:

Embed Image via LSB:

This option allows the user to embed an image into the currently opened image using the algorithm as described in section 5.2. Upon choosing the option, a dialog prompting the user to open a second image is displayed. The selected image must have the exact same pixel dimensions as the original or an error is displayed. If there was no error, the chosen image is embedded and a Save As dialog is displayed, prompting for a location to save the new stego-image. Note that the newly generated stego-image is not automatically loaded into the window. It must be reopened using the File -> Open dialog.

Embed Text via LSB:

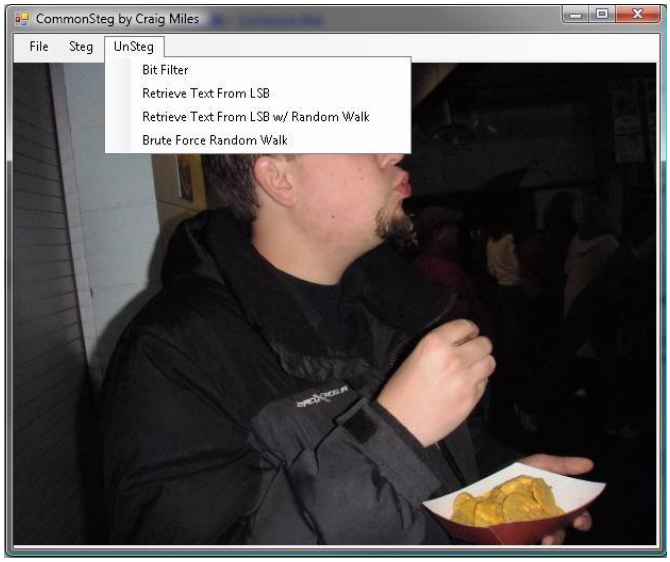
This option embeds ASCII text into the least significant bits of the image's pixels, starting from the most upper-left pixel, as described in section 5.3. Upon selection, the user is prompted with an input box for the text to be embedded into the image. Subsequently, a Save As dialog is displayed, prompting for a location to save the new stego-image.

Embed Text via LSB w/ Random Walk:

This option embeds ASCII text into a pseudo-random selection of the image's pixels as described in section 5.4. Upon selection, the user is prompted with input boxes for the text to be embedded into the image and a stego-key. The stego-key must be a positive

32-bit signed integer. Subsequently, a Save As dialog is displayed, prompting for a location to save the new stego-image.

UnSteg Menu



Bit Filter

The Bit Filter is used to recover an image message that was embedded using the Embed Image via LSB functionality. The least significant bits of the color intensity values for each pixel are shifted to the left, making them the most significant.

Retrieve Text From LSB

Retrieves text from the image that was embedded using the Embed Text via LSB functionality. It continues reading until it finds eight consecutive 1's in the LSBs. If the end of the image is reached before this occurs, an error is returned. Otherwise, the recovered message is displayed to the user.

Retrieve Text From LSB w/ Random Walk

Retrieves text from the image that was embedded using the Embed Text via LSB w/ Random Walk functionality. The user is prompted for the stego-key needed to determine the correct pixels from which to read the message. It continues reading until it finds eight consecutive 1's in the LSBs, then outputs the recovered message to the user.

Brute Force Random Walk

Generates a log of all ASCII messages embedded within the file for a user-inputted range of stego-keys. This takes a very long time for larger ranges of stego-keys.

Appendix D – CommonSteg Source Code

MainForm Class

```
/*
 * CommonSteg v2.1 by Craig Miles
 * Purpose: Implements several common steganographic techniques.
 * MainForm Class - Event handlers for Form actions.
 * Last Modified: May 23, 2010
 * */
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Collections;

namespace CommonSteg
{
    public partial class MainForm : Form
    {
        Bitmap bmp;

        /*************************************************************************
         *  MainForm() - Initializes the window.
         *  Arguments: NA
         *  Returns: NA
         **/
        public MainForm()
        {
            InitializeComponent();
        }


        /*************************************************************************
         *  openToolStripMenuItem_Click() - File -> Open event handler.
         *  Displays dialog to open an image file.  Displays image in pane.
         *  Arguments: NA
         *  Returns: NA
         **/
        private void openToolStripMenuItem_Click(object sender, EventArgs e)
        {
            // Displays an OpenFileDialog so the user can select a Cursor.
            OpenFileDialog openFileDialog1 = new OpenFileDialog();
            openFileDialog1.Filter = "Image Files|*.png;*.tif;*.gif;*.jpg";
            openFileDialog1.Title = "Select a Lossless Image File";

            // Show the Dialog.
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                // Assign the cursor in the Stream to the Form's Cursor property.
                bmp = new Bitmap(openFileDialog1.OpenFile());
                pictureBox1.Width = bmp.Width;
                pictureBox1.Height = bmp.Height;
                pictureBox1.Image = bmp;
                stegToolStripMenuItem.Enabled = true;
                unStegToolStripMenuItem.Enabled = true;
                SaveAsMenuItem1.Enabled = true;
            }
        }
```

```
        /*************************************************************************
         *  SaveAsMenuItem1_Click() - File -> Save As event handler.
         *  Provides dialog to save current image in pane to disk.
         *  Arguments: NA
         *  Returns: NA
         **/
        private void SaveAsMenuItem1_Click(object sender, EventArgs e)
        {
            SaveFileDialog saveFileDialog1 = new SaveFileDialog();
            saveFileDialog1.Filter = "PNG Image|*.png";
            saveFileDialog1.Title = "Save an Image File";
            saveFileDialog1.ShowDialog();
            if (saveFileDialog1.FileName != "")
            {
                System.IO.FileStream fs =
(System.IO.FileStream)saveFileDialog1.OpenFile();
                bmp.Save(fs, System.Drawing.Imaging.ImageFormat.Png);
            }
        }


        /*************************************************************************
         *  exitToolStripMenuItem_Click() - File -> Exit event handler.
         *  Terminates execution.
         *  Arguments: NA
         *  Returns: NA
         **/
        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }


        /*************************************************************************
         *  embedViaLSBToolStripMenuItem_Click() - Steg -> Embed Image via LSB event
handler.
         *  Only active after an image has been opened.  Prompts for the image to embed
and calls
         *      ImageSteg::EmbedImageIntoLSB
         *  Arguments: NA
         *  Returns: NA
         **/
        private void embedViaLSBToolStripMenuItem_Click(object sender, EventArgs e)
        {

            // Displays an OpenFileDialog so the user can select a Cursor.
            OpenFileDialog openFileDialog1 = new OpenFileDialog();
            openFileDialog1.Filter = "Image Files|*.png;*.tif;*.gif;*.jpg";
            openFileDialog1.Title = "Select a Lossless Image File To Embed";

            // Show the Dialog.
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                // Assign the cursor in the Stream to the Form's Cursor property.
                Bitmap ToEmbed = new Bitmap(openFileDialog1.OpenFile());
                ImageSteg Stego = new ImageSteg();
                Bitmap output = Stego.EmbedImageIntoLSB(bmp, ToEmbed);
                if (output != null)
                {
                    SaveFileDialog saveFileDialog1 = new SaveFileDialog();
                    saveFileDialog1.Filter = "PNG Image|*.png";
                    saveFileDialog1.Title = "Save an Image File";
                    saveFileDialog1.ShowDialog();
                    if (saveFileDialog1.FileName != "")
                    {
                        System.IO.FileStream fs =
(System.IO.FileStream)saveFileDialog1.OpenFile();
                        output.Save(fs, System.Drawing.Imaging.ImageFormat.Png);
                    }
```

```
                }
                else
                {
                    MessageBox.Show("Error: The image you wish to embed must have the
same pixel dimensions as the cover image.");
                }
            }
        }


        /*************************************************************************
         *  embedTextViaLSBToolStripMenuItem_Click_1() - Steg -> Embed Text via LSB event
handler.
         *  Only active after an image has been opened.  Prompts for the text to embed
and calls
         *      ImageSteg::EmbedTextIntoLSB
         *  Arguments: NA
         *  Returns: NA
         **/
        private void embedTextViaLSBToolStripMenuItem_Click_1(object sender, EventArgs e)
        {
            String Message = InputBox.Show("Input the text to embed into the cover image:
", "Textual Input");
            ImageSteg Stego = new ImageSteg();
            Bitmap output = Stego.EmbedTextIntoLSB(bmp, Message);
            SaveFileDialog saveFileDialog1 = new SaveFileDialog();
            saveFileDialog1.Filter = "PNG Image|*.png";
            saveFileDialog1.Title = "Save an Image File";
            saveFileDialog1.ShowDialog();
            if (saveFileDialog1.FileName != "")
            {
                System.IO.FileStream fs =
(System.IO.FileStream)saveFileDialog1.OpenFile();
                output.Save(fs, System.Drawing.Imaging.ImageFormat.Png);
            }
        }

        /*************************************************************************
         *  embedTextViaLSBWRandomWalkToolStripMenuItem_Click - Steg -> Embed Text via
LSB Random Walk event handler.
         *  Only active after an image has been opened.  Prompts for the text to embed
and seed (Int32), then calls
         *      ImageSteg::EmbedTextIntoLSBRandomWalk
         *  Arguments: NA
         *  Returns: NA
         **/
        private void embedTextViaLSBWRandomWalkToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            String Message = InputBox.Show("Input the text to embed into the cover image:
", "Textual Input");
            String seedStr = InputBox.Show("Input seed (any valid signed 32-bit
integer).", "Seed Input");
            Int32 seed = System.Convert.ToInt32(seedStr);
            ImageSteg Stego = new ImageSteg();
            Bitmap output = Stego.EmbedTextIntoLSBRandomWalk(bmp, Message, seed);
            if (output != null)
            {
                SaveFileDialog saveFileDialog1 = new SaveFileDialog();
                saveFileDialog1.Filter = "PNG Image|*.png";
                saveFileDialog1.Title = "Save an Image File";
                saveFileDialog1.ShowDialog();
                if (saveFileDialog1.FileName != "")
                {
                    System.IO.FileStream fs =
(System.IO.FileStream)saveFileDialog1.OpenFile();
                    output.Save(fs, System.Drawing.Imaging.ImageFormat.Png);
                }
            }
            else
            {
```

```
                MessageBox.Show("Image does not have enough pixels to hold the specified
message.");
            }
        }

        /*************************************************************************
         *  bitFilterToolStripMenuItem_Click() - UnSteg -> Bit Filter event handler.
         *  Calls ImageSteg::FilterLSB, updates image in pane with result.
         *  Arguments: NA
         *  Returns: NA
         **/
        private void bitFilterToolStripMenuItem_Click(object sender, EventArgs e)
        {
            ImageSteg Stego = new ImageSteg();
            pictureBox1.Image = Stego.FilterLSB(bmp, true);
        }



        /*************************************************************************
         *  retrToolStripMenuItem_Click() - UnSteg -> Retrieve Text from LSB event
handler.
         *  Calls ImageSteg::RetrieveTextFromLSB, outputs to user
         *  Arguments: NA
         *  Returns: NA
         **/
        private void retrToolStripMenuItem_Click(object sender, EventArgs e)
        {
            ImageSteg Stego = new ImageSteg();
            String output = Stego.RetrieveTextFromLSB(bmp);
            MessageBox.Show(output);
        }


        /*************************************************************************
         *  retrieveTextFromLSBWRandomWalkToolStripMenuItem_Click() - UnSteg -> Retrieve
Text from LSB Random Walk event handler.
         *  Prompts for seed, calls ImageSteg::RetrieveTextFromLSBRandomWalk, outputs
result to user
         *  Arguments: NA
         *  Returns: NA
         **/
        private void retrieveTextFromLSBWRandomWalkToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            ImageSteg Stego = new ImageSteg();
            String seedStr = InputBox.Show("Input seed (any valid signed 32-bit
integer).", "Seed Input");
            Int32 seed = System.Convert.ToInt32(seedStr);
            String output = Stego.RetrieveTextFromLSBRandomWalk(bmp, seed, true);
            MessageBox.Show(output);
        }


        /*************************************************************************
         *  bruteForceRandomWalkToolStripMenuItem_Click() - UnSteg -> Brute Force Random
Walk event handler.
         *  Calls ImageSteg::RetrieveTextFromLSBRandomWalk for a specified range of
seeds, and logs all valid textual messages.
         *  Arguments: NA
         *  Returns: NA
         **/
        private void bruteForceRandomWalkToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            Int32 low = 0;
            Int32 high = 100000000;
            ImageSteg Stego = new ImageSteg();
            System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch();
            watch.Start();
```

```
            ArrayList output = Stego.BruteForceRandomWalk(bmp, low, high); //2147483647
            watch.Stop();
                        String text = output.Count.ToString() + " Messages With Valid
ASCII Codes Were Found in " +
                                                (watch.ElapsedMilliseconds /
1000).ToString() + " Seconds\r\n" +
                                                "Seed Range: " + low.ToString() + " to " +
high.ToString() + "\r\n" +
                                                "----------------------------------------
-------\r\n";
            foreach (String s in output)
            {
                text = text + "\r\n" + s;
            }
            SaveFileDialog saveFileDialog1 = new SaveFileDialog();
            saveFileDialog1.Filter = "TXT File|*.txt";
            saveFileDialog1.Title = "Save the log file";
            saveFileDialog1.ShowDialog();
            if (saveFileDialog1.FileName != "")
            {
                System.IO.Stream fs = (System.IO.Stream)saveFileDialog1.OpenFile();
                System.IO.StreamWriter sw = new System.IO.StreamWriter(fs);
                sw.WriteLine(text);
                sw.Close();
                fs.Close();
            }
        }
    }
}
```

ImageSteg Class

```
/*
 * CommonSteg v2.1 by Craig Miles
 * Purpose: Implements several common steganographic techniques.
 * ImageSteg Class - Implements steganographic algorithms.
 * Last Modified: May 23, 2010
 * */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;
using System.Windows.Forms;
using System.Collections;


namespace CommonSteg
{
    class ImageSteg
    {
        Color embColor = new Color();
        Color oldColor = new Color();
        byte newR, newG, newB;


        /*************************************************************************
         *  EmbedImageIntoLSB()
         *  Embeds an image into LSBs of color pixels where pixel selection is sequential
         *  Arguments: Bitmap cover - image to hide text in
         *             Bitmap message - the image to hide
         *  Returns: Bitmap - Stego-Image containing message
         **/
        public Bitmap EmbedImageIntoLSB(Bitmap cover, Bitmap message)
        {
            Bitmap bmp = cover;
            Bitmap ToEmbed = message;
            if (ToEmbed.Height != bmp.Height || ToEmbed.Width != bmp.Width)
```

```
{
    return null;
}
else
{
    for (int ii = 0; ii <= bmp.Width - 1; ii++)
    {
        for (int jj = 0; jj <= bmp.Height - 1; jj++)
        {
            embColor = ToEmbed.GetPixel(ii, jj);
            oldColor = bmp.GetPixel(ii, jj);

            // RED
            if (embColor.R >= 192)
            {
                newR = SetLSBHigh(oldColor.R);
            }
            else if (embColor.R >= 128)
            {
                newR = SetLSBMid(oldColor.R);
            }
            else if (embColor.R >= 64)
            {
                newR = SetLSBLow(oldColor.R);
            }
            else
            {
                newR = SetLSBZero(oldColor.R);
            }

            // GREEN
            if (embColor.G >= 192)
            {
                newG = SetLSBHigh(oldColor.G);
            }
            else if (embColor.G >= 128)
            {
                newG = SetLSBMid(oldColor.G);
            }
            else if (embColor.G >= 64)
            {
                newG = SetLSBLow(oldColor.G);
            }
            else
            {
                newG = SetLSBZero(oldColor.G);
            }

            // BLUE
            if (embColor.B >= 192)
            {
                newB = SetLSBHigh(oldColor.B);
            }
            else if (embColor.B >= 128)
            {
                newB = SetLSBMid(oldColor.B);
            }
            else if (embColor.B >= 64)
            {
                newB = SetLSBLow(oldColor.B);
            }
            else
            {
                newB = SetLSBZero(oldColor.B);
            }

            Color newColor = System.Drawing.Color.FromArgb(newR, newG, newB);
            bmp.SetPixel(ii, jj, newColor);
        }
    }
    return bmp;
```

```
            }
        }

        /*************************************************************************
         *  EmbedTextIntoLSB()
         *  Embeds text into LSBs of color pixels where pixel selection is sequential
         *  Arguments: Bitmap cover - image to hide text in
         *             String message - the message to hide
         *  Returns: Bitmap - Stego-Image containing message
         **/
        public Bitmap EmbedTextIntoLSB(Bitmap cover, String message)
        {
            Bitmap bmp = cover;
            String ToEmbed = message;
            int x = 0;
            int y = 0;
            Color newColor;
            foreach (char c in ToEmbed)
            {
                for (int mask = 128; mask >= 1; mask = mask / 2)
                {
                    oldColor = cover.GetPixel(x, y);
                    if (((byte)c & mask) == mask)
                    {
                        newR = SetLSB1(oldColor.R);
                    }
                    else
                    {
                        newR = SetLSB0(oldColor.R);
                    }
                    //MessageBox.Show(newR.ToString());
                    newColor = System.Drawing.Color.FromArgb(newR, oldColor.G,
oldColor.B);
                    bmp.SetPixel(x, y, newColor);
                    if (x < bmp.Width - 1)
                    {
                        x++;
                    }
                    else
                    {
                        x = 0;
                        y++;
                    }
                }
            }
            for (int ii = 1; ii <= 8; ii++)
            {
                oldColor = cover.GetPixel(x, y);
                newR = SetLSB1(oldColor.R);
                //MessageBox.Show(newR.ToString());
                newColor = System.Drawing.Color.FromArgb(newR, oldColor.G, oldColor.B);
                bmp.SetPixel(x, y, newColor);
                if (x < bmp.Width - 1)
                {
                    x++;
                }
                else
                {
                    x = 0;
                    y++;
                }
            }
            return bmp;
        }

        /*************************************************************************
         *  EmbedTextIntoLSBRandomWalk()
         *  Embeds text into LSBs of color pixels where pixel selection is handled via
PRNG
         *  Arguments: Bitmap cover - image to hide text in
         *             String message - the message to hide
```

```
         *             Int32 seed - Seed for PRNG
         *  Returns: Bitmap - Stego-Image containing message
         **/
        public Bitmap EmbedTextIntoLSBRandomWalk(Bitmap cover, String message, Int32
seed)
        {
            Bitmap bmp = cover;
            String ToEmbed = message;
            int x = 0;
            int y = 0;
            int curPixel;
            int totalPixels = bmp.Width * bmp.Height;
            if (totalPixels < ((message.Length * 8) + 8))
            {
                return null;
            }
            Color newColor;
            Random rnd = new Random(seed);
            curPixel = rnd.Next(0, totalPixels - 1);
            x = curPixel % bmp.Width;
            y = curPixel / bmp.Width;

            foreach (char c in ToEmbed)
            {
                for (int mask = 128; mask >= 1; mask = mask / 2)
                {
                    oldColor = cover.GetPixel(x, y);
                    if (((byte)c & mask) == mask)
                    {
                        newR = SetLSB1(oldColor.R);
                    }
                    else
                    {
                        newR = SetLSB0(oldColor.R);
                    }
                    //MessageBox.Show(newR.ToString());
                    newColor = System.Drawing.Color.FromArgb(newR, oldColor.G,
oldColor.B);
                    bmp.SetPixel(x, y, newColor);
                    curPixel = rnd.Next(0, totalPixels - 1);
                    x = curPixel % bmp.Width;
                    y = curPixel / bmp.Width;
                }
            }
            for (int ii = 1; ii <= 8; ii++)
            {
                oldColor = cover.GetPixel(x, y);
                newR = SetLSB1(oldColor.R);
                //MessageBox.Show(newR.ToString());
                newColor = System.Drawing.Color.FromArgb(newR, oldColor.G, oldColor.B);
                bmp.SetPixel(x, y, newColor);
                curPixel = rnd.Next(0, totalPixels - 1);
                x = curPixel % bmp.Width;
                y = curPixel / bmp.Width;
            }
            return bmp;
        }

        /*************************************************************************
         *  FilterLSB()
         *  Sets all but the 2 LSBs for all pixel RGBs to 0.  Shifts them to MSBs if
         *      amplify is set.
         *  Arguments: Bitmap stego - image to filter
         *             Boolean amplify - Amplify LSBs?
         *  Returns: Bitmap - Filtered image.
         **/
        public Bitmap FilterLSB(Bitmap stego, Boolean amplify)
        {
            Bitmap bmp = stego;
            Color newColor = new Color();
```

```
            for (int ii = 0; ii <= bmp.Width - 1; ii++)
            {
                for (int jj = 0; jj <= bmp.Height - 1; jj++)
                {
                    oldColor = bmp.GetPixel(ii, jj);
                    if (amplify)
                    {
                        newColor = System.Drawing.Color.FromArgb((oldColor.R & 3) << 6,
(oldColor.G & 3) << 6, (oldColor.B & 3) << 6);
                    }
                    else
                    {
                        newColor = System.Drawing.Color.FromArgb(oldColor.R & 3,
oldColor.G & 3, oldColor.B & 3);
                    }
                    bmp.SetPixel(ii, jj, newColor);
                }
            }
            return bmp;
        }

        /*************************************************************************
         *  RetrieveTextFromLSB()
         *  Retrieves a message from an image via sequential starting pixels.
         *  Arguments: Bitmap stego - image to extract text from
         *  Returns: String - Recovered message
         **/
        public String RetrieveTextFromLSB(Bitmap stego)
        {
            byte c = 0;
            StringBuilder builder = new StringBuilder();
            String output = "";
            int x = 0;
            int y = 0;
            Color pixelColor;


            while (c != 255)
            {
                c = 0;
                for (int ii = 7; ii >= 0; ii--)
                {
                    pixelColor = stego.GetPixel(x, y);
                    if (pixelColor.R % 2 != 0)
                    {
                        c += (byte)Math.Pow(2, ii);
                    }
                    if (x < stego.Width - 1)
                    {
                        x++;
                    }
                    else
                    {
                        x = 0;
                        y++;
                    }
                }
                if (c != 255)
                {
                    builder.Append((char)c);
                }
            }

            output = builder.ToString();
            return output;
        }

        /*************************************************************************
         *  RetrieveTextFromLSBRandomWalk()
         *  Retrieves a message from an image via Random Walk algorithm.  If
WorkWithGarbage
```

```
         *      set, will return non-valid ASCII strings.
         *  Arguments: Bitmap stego - image to extract text from
         *             Int32 seed - Seed for PRNG
         *             Boolean WorkWithGarbage - Validate output?
         *  Returns: String - Detected message
         **/
        public String RetrieveTextFromLSBRandomWalk(Bitmap stego, Int32 seed, Boolean
WorkWithGarbage)
        {
            byte c = 0;
            StringBuilder builder = new StringBuilder();
            String output = "";
            int x = 0;
            int y = 0;
            Color pixelColor;
            int curPixel;
            int totalPixels = stego.Width * stego.Height;
            Random rnd = new Random(seed);
            curPixel = rnd.Next(0, totalPixels - 1);
            x = curPixel % stego.Width;
            y = curPixel / stego.Width;

            while (c != 255)
            {
                c = 0;
                for (int ii = 7; ii >= 0; ii--)
                {
                    pixelColor = stego.GetPixel(x, y);
                    if (pixelColor.R % 2 != 0)
                    {
                        c += (byte)Math.Pow(2, ii);
                    }
                    curPixel = rnd.Next(0, totalPixels - 1);
                    x = curPixel % stego.Width;
                    y = curPixel / stego.Width;
                }
                if (!WorkWithGarbage)
                {
                    if ((int)c < 32 || ((int)c > 126 && (int)c != 255))
                    {
                        return null;
                    }
                }
                if (c != 255)
                {
                    builder.Append((char)c);
                }
            }

            output = builder.ToString();
            return output;
        }

        /*************************************************************************
         *  BruteForceRandomWalk()
         *  Extracts all ASCII message strings from image in user specified seed range
         *  Arguments: Bitmap stego - image to attack
         *             Int32 lowSeed - starting seed to BF from
         *             Int32 highSeed - max seed to BF to
         *  Returns: ArrayList<String> - Log of all found ASCII messages
         **/
        public ArrayList BruteForceRandomWalk(Bitmap stego, Int32 lowSeed, Int32
highSeed)
        {
            String text;
            ArrayList returnedStrings = new ArrayList();
            for(int ii = lowSeed; ii <= highSeed; ii++)
            {
                text = RetrieveTextFromLSBRandomWalk(stego, ii, false);
                if (text != null && text.CompareTo("") != 0)
                {
```

```
                text = "Seed: " + ii.ToString() + " Message: " + text;
                returnedStrings.Add(text);
            }
        }
        return returnedStrings;
    }

    /*************************************************************************
     *  SetLSB1()
     *  Sets the least significant bit of a byte to 1.
     *  Arguments: byte input - byte to modify
     *  Returns: byte - modified byte
     **/
    private byte SetLSB1(byte input)
    {
        byte output = input;
        if ((input & 1) != 1)
        {
            output = (byte)(output ^ 1);
        }
        return output;
    }

    /*************************************************************************
     *  SetLSB0()
     *  Sets the least significant bit of a byte to 0.
     *  Arguments: byte input - byte to modify
     *  Returns: byte - modified byte
     **/
    private byte SetLSB0(byte input)
    {
        byte output = input;
        if ((input & 1) == 1)
        {
            output = (byte)(output ^ 1);
        }
        return output;
    }


    /*************************************************************************
     *  SetLSBHigh()
     *  Sets the 2 least significant bits of a byte to 11.
     *  Arguments: byte input - byte to modify
     *  Returns: byte - modified byte
     **/
    private byte SetLSBHigh(byte input)
    {
        byte output = input;
        if ((input & 2) != 2)
        {
            output = (byte)(input ^ 2);
        }
        if ((input & 1) != 1)
        {
            output = (byte)(output ^ 1);
        }
        return output;
    }

    /*************************************************************************
     *  SetLSBMid()
     *  Sets the 2 least significant bits of a byte to 10.
     *  Arguments: byte input - byte to modify
     *  Returns: byte - modified byte
     **/
    private byte SetLSBMid(byte input)
    {
        byte output = input;
        if ((input & 2) != 2)
        {
```

```
                output = (byte)(input ^ 2);
            }
            if ((input & 1) == 1)
            {
                output = (byte)(output ^ 1);
            }
            return output;
        }

        /*************************************************************************
         *  SetLSBLow()
         *  Sets the 2 least significant bits of a byte to 01.
         *  Arguments: byte input - byte to modify
         *  Returns: byte - modified byte
         **/
        private byte SetLSBLow(byte input)
        {
            byte output = input;
            if ((input & 2) == 2)
            {
                output = (byte)(input ^ 2);
            }
            if ((input & 1) != 1)
            {
                output = (byte)(output ^ 1);
            }
            return output;
        }

        /*************************************************************************
         *  SetLSBZero()
         *  Sets the 2 least significant bits of a byte to 00.
         *  Arguments: byte input - byte to modify
         *  Returns: byte - modified byte
         **/
        private byte SetLSBZero(byte input)
        {
            byte output = input;
            if ((input & 2) == 2)
            {
                output = (byte)(input ^ 2);
            }
            if ((input & 1) == 1)
            {
                output = (byte)(output ^ 1);
            }
            return output;
        }
    }
}
```