JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY
JUIT
विद्या तत्व ज्योतिसम:

Jaypee University of Information Technology
Solan (H.P.)
LEARNING RESOURCE CENTER

Acc. Num. SP03094 Call Num:

General Guidelines:

- Library books should be used with great care.
- Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT
SP03094

STUDY & IMPLEMENTATION OF STEGANOGRAPHIC METHODS IN IMAGE FILES

BY

MAYANK AGARWAL (031225)
NAKUL JINDAL (031205)

Learning Resource Centre
JUIT
Waknaghat, Solan (H.P.)

May 2007

Submitted in partial fulfillment of the Degree of Bachelor of Technology
Department of Computer Science Engineering and Information Technology,
Jaypee University of Information Technology -Waknaghat

CERTIFICATE

This is to certify that the work entitled, "Study & Implementation of Steganographic Methods in Image Files" submitted by Mayank Agarwal & Nakul Jindal in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Supervisor:

[illegible]

Mr. S.P.Ghrera

HOD

Department of Computer Science Engineering and Information Technology,

Jaypee University of Information Technology,

Waknaghat, Solan – 173215, Himachal Pradesh,

INDIA

ACKNOWLEDGEMENT

We wish to express our earnest gratitude to **Mr. S. P. Ghrera,** for providing us invaluable guidance and timely suggestions by the help of which we successfully completed our project. We'd also like to thank him for his moral support in times when the project was losing pace.

We would like to the thank the faculty of the Computer Science and Engineering Department of Jaypee University of Information Technology, Waknaghat for their valuable suggestions that made helped us improve our project.

We would like to thank all the staff members of the Computing facilities of **Jaypee University of Information Technology**, Waknaghat, for providing us with support and facilities required for the completion of this project.

ABSTRACT

A study of the various Steganographic Methods for image files has been done by us. Report of this study is presented. One of the studied techniques was implemented, the details of which are presented.

TABLE OF CONTENTS

Chapter 4 The Project

LIST OF FIGURES

CHAPTER 1
STEGANOGRAPHY

Introduction to Information Hiding

As audio, video, and other works become available in digital form, the ease with which perfect copies can be made, may lead to large-scale unauthorized copying which might undermine the music, film, book, and software publishing industries. These concerns over protecting copyright have triggered significant research to find ways to hide copyright messages and serial numbers into digital media; the idea is that the latter can help to identify copyright violators, and the former to prosecute them. Thus came the need for Information Hiding. The need to embed information into such data arised and gave birth to digital watermarking. Watermarking was already in existence at that time.

At the same time, moves by various governments to restrict the availability of encryption services have motivated people to study methods by which private messages can be embedded in seemingly innocuous cover messages. Techniques used in digital watermarking and other techniques were used so that data could be hidden in these cover messages. The field dealing with these techniques is called steganography.

Steganography is an important sub discipline of information hiding. While cryptography is about protecting the content of messages, steganography is about concealing their very existence. This modern adaptation of steganographia (Trithemius, 1462–1516), assumed from Greek στεγανός. γράφειν literally means "covered writing" [1], and is usually interpreted to mean hiding information in other information.

1.1 A Brief History of Information Hiding (Steganography)

- The first description of the use of steganography dates back to the Greeks. Herodotus [2] tells how a message was passed to the Greeks about Xerses' hostile intentions underneath the wax of a writing tablet, and describes a technique of dotting successive letters in a cover text with a secret ink, due to Aeneas the Tactician.

- Pirate legends tell of the practice of tattooing secret information, such as a map, on the head of someone, so that the hair would conceal it.
- Kahn tells of a trick used in China of embedding a code ideogram at a prearranged position in a dispatch; a similar idea led to the grille system used in medieval Europe, where a wooden template would be placed over a seemingly innocuous text, highlighting an embedded secret message.
- During WWII the grille method or some variants were used by spies. In the same period, the Germans developed microdot technology, which prints a clear, good quality photograph shrinking it to the size of a dot.
- More obscurely, during World War II, a spy for the Japanese in New York City, Velvalee Dickinson, sent information to accommodation addresses in neutral South America. She was a dealer in dolls, and her letters discussed how many of this or that doll to ship. The stegotext in this case was the doll orders; the 'plaintext' being concealed was itself a codetext giving information about ship movements, etc. Her case became somewhat famous and she became known as the Doll Woman.
- There are rumors that during the 1980's Margareth Thatcher, then Prime Minister in UK, became so irritated about press leaks of cabinet documents, that she had the word processors programmed to encode the identity of the writer in the word spacing, thus being able to trace the disloyal ministers.
- During the "Cold War" period, US and USSR wanted to hide their sensors in the enemy's facilities. These devices had to send data to their nations, without being spotted.

Today, steganography is researched both for legal and illegal reasons.

- Among the first ones there is war telecommunications, which use spread spectrum or meteor scatter radio in order to conceal both the message and its source.

- In the industry market, with the advent of digital communications and storage, one of the most important issues is copyright enforcement, so digital watermarking techniques are being developed to restrict the use of copyrighted data.
- Another important use is to embed data about medical images, so that there are no problems with matching patient's records and images.
- Among illegal ones is the practice of hiding strongly-encrypted data to avoid controls by cryptography export laws.

1.2 Wisdom from Cryptography

Although steganography is different from cryptography, we can borrow many of the techniques and much practical wisdom from the latter, a more thoroughly researched discipline.

*In 1883, **Auguste Kerckhoffs** enunciated the first principles of cryptographic engineering, in which he advises that we assume the method used to encipher data is known to the opponent, so security must lie only in the choice of key* [3]

Applying this wisdom, we obtain a tentative definition of a secure stego-system: one where an opponent who understands the system, but does not know the key, can obtain no evidence (or even grounds for suspicion) that a communication has taken place. It will remain a central principle that steganographic processes intended for wide use should be published, just like commercial cryptographic algorithms and protocols.

So one might expect that designers of copyright marking systems would publish the mechanisms they use, and rely on the secrecy of the keys employed. Sadly, this is not the case; many purveyors of such systems keep their mechanisms subject to nondisclosure agreements, sometimes offering the rationale that a patent is pending.

That any of these security-by-obscurity systems ever worked was a matter of luck. Yet many steganographic systems available today just embed the "hidden" data in the least significant bits of an audio or video file—which is trivial for a capable opponent to detect and remove.

1.3 Principles of Steganography

The "classic" model for invisible communication was first proposed by Simmons [4] as the "prisoners' problem." Alice[1] and Bob are arrested for some crime and are thrown in two different cells. They want to develop an escape plan, but unfortunately all communications between each other are arbitrated by a warden named Wendy. She will not let them communicate through encryption and if she notices any suspicious communication, she will place them in solitary confinement and thus suppress the exchange of all messages. So both parties must communicate invisibly in order not to arouse Wendy's suspicion; they have to set up a *subliminal channel.* A practical way to do so is to hide meaningful information in some harmless message: Bob could, for instance, create a picture of a blue cow lying on a green meadow and send this piece of modern art to Alice. Wendy has no idea that the colors of the objects in the picture transmit information.

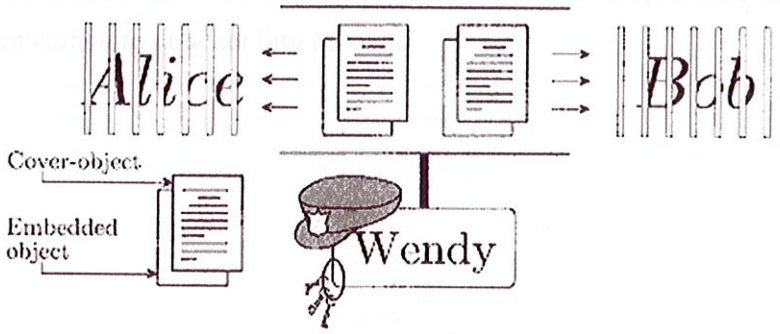


Fig.1-1 The Prisoner's Problem [5]

[1] In the field of cryptography, communication protocols usually involve two fictional characters named Alice and Bob. The standard convention is to name the participants in the protocol

alphabetically (Carol and Dave often succeed Alice and Bob in a multiperson protocol), or with a name whose first character matches the first letter of their role (e.g., Wendy the warden).

Unfortunately there are other problems which may hinder the escape of Alice and Bob. Wendy may alter the message Bob has sent to Alice. For example, she could change the color of Bob's cow to red, and so destroy the information; she then acts as an *active warden.* Even worse, if she acts in a *malicious* way, she could forge messages and send a message to one of the prisoners through the subliminal channel while pretending to be the other.

The above model is generally applicable to many situations in which invisible communication—*steganography*— takes place. Alice and Bob represent two communication parties, wanting to exchange secret information invisibly. The warden Wendy represents an eavesdropper who is able to read and probably alter messages sent between the communication partners (see figure 1-1).

Whereas cryptographic techniques try to conceal the contents of a message, steganography goes yet a bit further: it tries to hide the fact that a communication even exists. Two people can communicate covertly by exchanging unclassified messages containing confidential information. Both parties have to take the presence of a *passive, active* or even *malicious* attacker into account.

1.4 Frameworks for Secret Communication

Most applications of steganography follow one general principle, illustrated in figure 1-2 Alice, who wants to share a secret message m with Bob, randomly chooses (using the private random source r) a harmless message c, called cover-object, which can be transmitted to Bob without raising suspicion, and embeds the secret message into c, probably by using a key k, called stego-key. Alice therefore changes the cover c to a stego-object s. This must be done in a very careful way, so that a third party, knowing only the apparently harmless message s, cannot detect the existence of the secret. In a ''perfect" system, a normal cover should not be distinguishable from a stego-object, neither by a human nor by a computer looking for statistical pattern. Theoretically, covers could be any computer-readable data such as image files, digital sound, or written text.

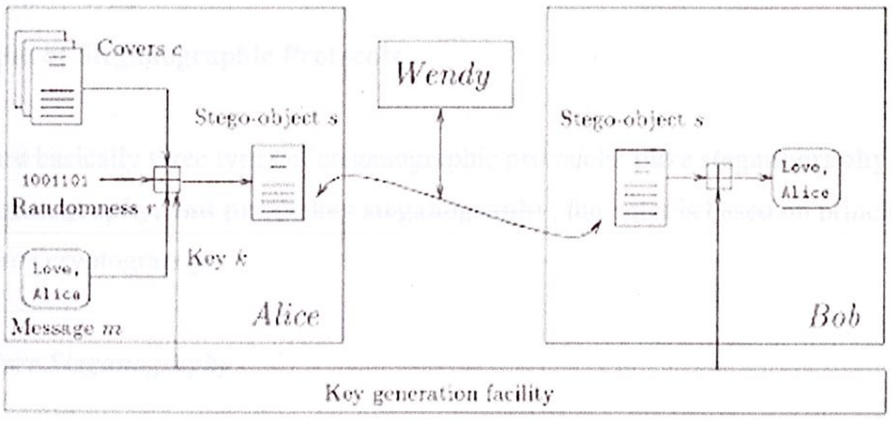


Fig.1-2 Schematic description of Steganography: Alice randomly chooses a cover c using her private random source r and embeds the message m in c using a key k, creating the stego-object s which she passes on to Bob. Bob reconstructs m with the key k he shares with Alice.

Alice then transmits s over an insecure channel to Bob and hopes that Wendy will not notice the embedded message. Bob can reconstruct m since he knows the embedding method used by Alice and has access to the key k used in the embedding process. This extraction process should be possible without the original cover c.

Thus, the security of invisible communication lies mainly in the inability to distinguish cover-objects from stego-objects.

In practice however, not all data can be used as cover for secret communication, since the modifications employed in the embedding process should not be visible to anyone not involved in the communication process. This fact requires the cover to contain sufficient redundant data, which can be replaced by secret information. In fact, it turns out that noisy data has more advantageous properties in most steganographic applications. Obviously a cover should never be used twice, since an attacker who has access to two "versions" of one cover can easily detect and possibly reconstruct the message. To avoid accidental reuse, both sender and receiver should destroy all covers they have already used for information transfer.

1.5 Types of Steganographic Protocols

There are basically three types of steganographic protocols: **pure steganography, secret key steganography, and public key steganography**; the latter is based on principles of public key cryptography.

1.5.1 Pure Steganography

We call a steganographic system which does not require the prior exchange of some secret information (like a stego-key) pure steganography. Formally, the embedding process can be described as a mapping $E: C \times M \rightarrow C$, where C is the set of possible covers and M the set of possible messages. The extraction process consists of a mapping $D : C \rightarrow M$, extracting the secret message out of a cover. Clearly, it is necessary that $|C| \geq |M|$. Both sender and receiver must have access to the embedding and extraction algorithm, but the algorithms should not be public.

1.5.2 Secret Key Steganography

With pure steganography, no information (apart from the functions E and D) is required to start the communication process; the security of the system thus depends entirely on its secrecy. This is not very secure in practice because this violates Kerckhoffs' principle (as stated previously). So we must assume that Wendy knows the algorithm Alice and Bob use for information transfer. In theory, she is able to extract information out of every cover sent between Alice and Bob. The security of a steganographic system should thus rely on some secret information traded by Alice and Bob, the stego-key. Without knowledge of this key, nobody should be able to extract secret information out of the cover.

A secret key steganography system is similar to a symmetric cipher: the sender chooses a cover c and embeds the secret message into c using a secret key k. If the key used in the embedding process is known to the receiver, he can reverse the process and extract the secret message. Anyone who does not know the secret key should not be able to obtain evidence of the encoded information. Again, the cover c and the stego-object can be perceptually similar.

1.5.3 Public Key Steganography

As in public key cryptography, public key steganography does not rely on the exchange of a secret key. Public key steganography systems require the use of two keys, one private and one public key; the public key is stored in a public database. Whereas the public key is used in the embedding process, the secret key is used to reconstruct the secret message.

One way to build a public key steganography system is the use of a public key cryptosystem. We will assume that Alice and Bob can exchange public keys of some public key cryptography algorithm before imprisonment (this is, however, a more reasonable assumption). Public key steganography utilizes the fact that the decoding function *D* in a steganography system can be applied to any cover *c,* whether or not it already contains a secret message (recall that D is a function on the entire set C). In the latter case, a random element of M will be the result, we will call it "natural randomness" of the cover. If one assumes that this natural randomness is statistically indistinguishable from ciphertext produced by some public key cryptosystem, a secure steganography system can be built by embedding ciphertext rather than unencrypted secret messages.

A protocol which allows public key steganography has been proposed by Anderson in [6, 7]; it relies on the fact that encrypted information is random enough to "hide in plain sight": Alice encrypts the information with Bob's public key to obtain a random-looking message and embeds it in a channel known to Bob (and hence also to Wendy), thereby replacing some of the "natural randomness" with which every communication process is accompanied. We will assume that both the cryptographic algorithms and the embedding functions are publicly known. Bob, who cannot decide a priori if secret information is transmitted in a specific cover, will suspect the arrival of a message and will simply try to extract and decrypt it using his private key. If the cover actually contained information, the decrypted information is Alice's message.

Since we assumed that Wendy knows the embedding method used, she can try to extract the secret message sent from Alice to Bob. However, if the encryption method produces random-looking ciphertext, Wendy will have no evidence that the extracted information is more than some random bits. She thus cannot decide if the extracted information is meaningful or just part of the natural randomness, unless she is able to break the cryptosystem.

1.6 Security of Steganography Systems

Although breaking a steganography system normally consists of three parts: detecting, extracting, and disabling embedded information, a system is already insecure if an attacker is able to prove the existence of a secret message. In developing a formal security model for steganography we must assume that an attacker has unlimited computation power and is able and willing to perform a variety of attacks. If he cannot confirm his hypothesis that a secret message is embedded in a cover, then a system is theoretically secure.

1.6.1 Perfect Security

Cachin [8] gave a formal information-theoretic definition of the security of steganographic systems. The main idea is to refer to the selection of a cover as a random variable *C* with probability distribution *P_C*. The embedding of a secret message can be seen as a function defined in *C;* let *P_S* be the probability distribution of *$E_K(c, m, k)$,* that is the set of all stego-objects produced by the steganographic system. If a cover *c* is never used as a stego-object, then *$P_S(c) = 0$*. In order to calculate *P_S,* probability distributions on *K* and *M* must be imposed. Using the definition of the relative entropy *$D(P_1||P_2)$* between two distributions *P_1* and *P_2* defined on the set *Q,*

$$D(P_1 \| P_2) = \sum_{q \in Q} P_1(q) \log_2 \frac{P_1(q)}{P_2(q)}$$

—which measures the inefficiency of assuming that the distribution is *P_2* where the true distribution is *P_1*—the impact of the embedding process on the distribution *P_C* can be measured. Specifically, we define the security of a steganography system in terms of *$D(P_C||P_S)$:*

(Perfect security) *Let A be a steganography system, Ps the probability distribution of the stegocovers sent via the channel, and P_C the probability distribution of C.A is called ε - secure against passive attackers, if*

$$D(P_C \| P_S) \leq \varepsilon$$

—which measures the inefficiency of assuming that the distribution is *P_2* where the true distribution is *P_1*—the impact of the embedding process on the distribution *P_C* can be measured.

1.6.2 Detecting Secret Messages

A passive attacker (Wendy) has to decide whether a cover c sent from Bob to Alice contains secret information or not. This task can be formalized as a statistical hypothesis-testing problem. Therefore, Wendy defines a test function f: C → {0, 1}:

$$f(c) = \begin{cases} 1 & c \text{ contains a secret message} \\ 0 & \text{otherwise} \end{cases}$$

which Wendy uses to classify covers as they are passed on via the insecure channel. In some cases Wendy will correctly classify the cover; in other cases she will not detect a hidden message, making a type-II error. It is also possible that Wendy falsely detects a hidden message in a cover which does not contain information; she then makes a type-I error. Practical steganography systems try to maximize the probability β that a passive attacker makes a type-II error. An ideal system would have $\beta = 1$.

1.7 Information Hiding in Noisy Data

As we know, steganography utilizes the existence of redundant information in a communication process. Images or digital sound naturally contain such redundancies in the form of a noise component. We will assume without loss of generality that the cover *c* can be represented by a sequence of binary digits. In the case of a digital sound this sequence is just the sequence of samples over time; in the case of a digital image, a sequence can be obtained by vectorizing the image (i.e., by lining up the grayscale or color values in a left-to-right and top-to-bottom order). Let *l(c)* be the number of elements in the sequence, *m* the secret message, and *l(m)* its length in bits.

The general principle underlying most steganographic methods is to place the secret message in the noise component of a signal. If it is possible to code the information in such a way that it is indistinguishable from true random noise, an attacker has no chance in detecting the secret communication.

The simplest way of hiding information in a sequence of binary numbers is replacing the least significant bit (LSB) of every element with one bit of the secret message m. In floating point arithmetic, the least significant bit of the mantissa can be used instead. Since normally the size of the hidden message is much less than the number of bits available to hide the information (l(m) << l(c)) the rest of the LSB can be left unchanged. Since flipping the LSB of a byte (or a word) only means the addition or subtraction of a small quantity, the sender assumes that the difference will lie within the noise range and that it will therefore not be generally noticed. Obviously this technique does not provide a high level of security. An attacker can simply try to "decode" the cover, just as if he were the receiver. In addition, the algorithm changes the statistical properties of the cover significantly, even if the message consists of truly random bits.

This technique can be improved. Instead of using every cover-element for information transfer, it is possible to select only some elements in a rather random manner according to a secret key and leave the others unchanged. This selection can be done using a pseudorandom number generator; [9] report a system in which the output of the random number generator is used to spread the sequence of message bits over the cover by determining the number of cover-elements which are left unchanged between two elements used for information transfer.

1.8 Active and Malicious Attackers

Active attackers are able to change a cover during the communication process; It is a general assumption that an active attacker is not able to change the cover and its semantics entirely, but only make minor changes so that the original and the modified cover-object stay perceptually or semantically similar. An attacker is malicious if he forges messages or starts steganography protocols under the name of one communication partner.

1.8.1 Active Attackers: Robust Steganography

Steganographic systems are extremely sensitive to cover modifications, such as image processing techniques (like smoothing, filtering, and image transformations) in the case of digital images and filtering in the case of digital sound. But even a lossy compression can result in total information loss. Lossy compression techniques try to reduce the amount of information by removing imperceptible signal components and so often remove the secret information which has previously been added.

An active attacker, who is not able to extract or prove the existence of a secret message, thus can simply add random noise to the transmitted cover and so try to destroy the information. In the case of digital images, an attacker could also apply image processing techniques or convert the image to another file format. All of these techniques can be harmful to the secret communication. Another practical requirement for a steganography system therefore is robustness. A system is called robust if the embedded information cannot be altered without making drastic changes to the stego-object.

It should be clear that there is a trade-off between security and robustness. The more robust a system will be against modifications of the cover, the less secure it can be, since robustness can only be achieved by redundant information encoding which will degrade the cover heavily and possibly alter the probability distribution P_S.

Generally, there are two approaches in making steganography robust. First, by foreseeing possible cover modifications [10, 11] the embedding process itself can be made robust so that modifications will not entirely destroy secret information. A second approach tries to reverse the modifications that have been applied by the attacker to the cover, so that the original stego-object can be restored.

1.8.2 Supraliminal Channels

If we assume that an active attacker can only make minor changes to a stego-object, then every cover contains some sort of perceptually significant information which cannot be removed without entirely changing the semantics of the cover. By encoding a secret message in a way that it forms such a perceptually significant part, information can be transmitted between two communication partners with high integrity. Craver [12] calls such a channel a *supraliminal channel:* "information is hidden in plain sight, so obviously, in fact, that it is impossible to modify without gross modifications to the transmitted objects."

The covers used for secret communication can be described by a *cover-plot,* a formal description of the perceptually significant parts of the cover. Let **S** be the set of all cover-plots and f be a function f: $\mathbf{S} \rightarrow \{0, 1\}^N$, called *cover-plot function.* To embed a bitstring $x \in \{0, 1\}^N$ in a supraliminal channel, Alice chooses one element $s \in f_{-1}(x)$ and sends a cover conforming to the cover-plot s over the insecure channel. Wendy probably suspects the use of a subliminal channel and changes the cover slightly in an attempt to remove secret messages encoded in the noise component, but is not able to change the cover-plot. Bob reconstructs the cover-plot s out of the cover he received and applies the function f in order to recover x; see Figure

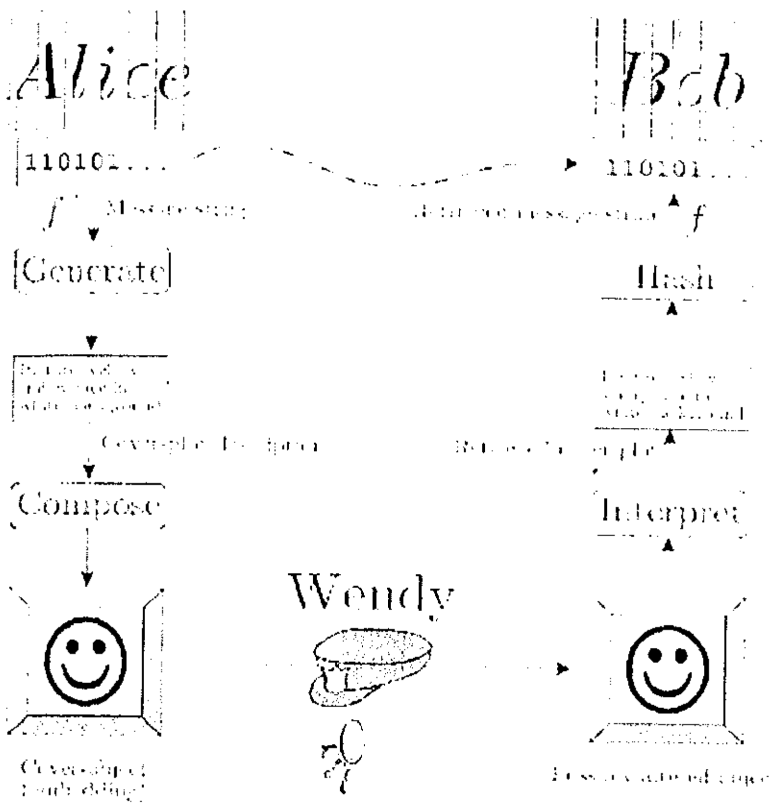


Fig.1-3 Schematic description of a supraliminal channel.

1.8.3 Malicious Attackers: Secure Steganography

In the presence of a malicious attacker, robustness is not enough. If the embedding method is not dependent on some secret information shared by sender and receiver, (i.e., in the case of pure steganography or public key steganography) an attacker can forge messages, since the recipient is not able to verify the correctness of the sender's identity. Thus, to avoid such an attack, the algorithm must be robust and secure. We can define a *secure steganographic algorithm* in terms of four requirements:

- Messages are hidden using a public algorithm and a secret key; the secret key must identify the sender uniquely;
- Only a holder of the correct key can detect, extract, and prove the existence of the hidden message. Nobody else should be able to find any statistical evidence of a message's existence;
- Even if the enemy knows (or is able to select) the contents of one hidden message, he should have no chance of detecting others;
- It is computationally infeasible to detect hidden messages.

References

[1] Murray, A. H., and R. W. Burchfiled (eds.), *The Oxford English dictionary: being a corrected re-issue,* Oxford, England: Clarendon Press, 1933.

[2] Herodotus, *The Histories,* London, England: J. M. Dent & Sons, Ltd, 1992.

[3] Kerckhoffs, A., "La Cryptographie Militaire," *Journal des Sciences Militaires,* vol. 9, Jan. 1883, pp. 5–38.

[4] Simmons, G. J., "The Prisoners' Problem and the Subliminal Channel," in Advances in Cryptology, Proceedings of CRYPTO '83, Plenum Press, 1984, pp. 51–67.

[5] Craver, S., "On Public-Key Steganography in the Presence of an Active Warden," Technical Report RC 20931, IBM, 1997.

[6] Anderson, R. J., "Stretching the Limits of Steganography," in Information Hiding: First International Workshop, Proceedings, vol. 1174 of Lecture Notes in Computer Science, Springer, 1996, pp. 39–48.

[7] Anderson, R. J., and F. A. P. Petitcolas, "On The Limits of Steganography," IEEE Journal of Selected Areas in Communications, vol. 16, no. 4, 1998, pp. 474–481.

[8] Cachin, C., "An Information-Theoretic Model for Steganography," in Proceedings of the Second International Workshop on Information Hiding, vol. 1525 of Lecture Notes in Computer Science, Springer, 1998, pp. 306–318.

[9] Möller, S., A. Pfitzmann, and I. Stirand, "Computer Based Steganography: How It Works and Why Therefore Any Restrictions on Cryptography Are Nonsense, At Best," in Information Hiding: First International Workshop, Proceedings, vol. 1174 of Lecture Notes in Computer Science, Springer, 1996, pp. 7–21.

[10] Johnson, N. F., Z. Duric, and S. Jajodia, "A Role for Digital Watermarking in Electronic Commerce," to appear in *ACM Computing Surveys.*

[11] Johnson, N. F., "An Introduction to Watermark Recovery from Images," in *SANS Intrusion Detection and Response Conference, Proceedings,* 1999.

[12] Craver, S., "On Public-Key Steganography in the Presence of an Active Warden," Technical Report RC 20931, IBM, 1997.

CHAPTER 2
A SURVEY OF STEGANAGRAPHIC TECHNIQUES

Types of Steganographic Techniques

There are several approaches in classifying steganographic systems. One could categorize them according to the type of covers used for secret communication. A classification according to the cover modifications applied in the embedding process is another possibility. We want to follow the second approach and group steganographic methods in six categories, although in some cases an exact classification is not possible:

• **Substitution systems** substitute redundant parts of a cover with a secret message;

• **Transform domain** techniques embed secret information in a transform space of the signal (e.g., in the frequency domain);

• **Spread spectrum** techniques adopt ideas from spread spectrum communication;

• **Statistical methods** encode information by changing several statistical properties of a cover and use hypothesis testing in the extraction process;

• **Distortion techniques** store information by signal distortion and measure the deviation from the original cover in the decoding step;

• **Cover generation methods** encode information in the way a cover for secret communication is created.

Notations:

- Any cover can be represented by a sequence of numbers c_i of length $l(c)$ (i.e., $1 \leq i \leq l(c)$).
- The stego-object is denoted by s which is again a sequence s_i of length $l(c)$.
- To index all cover-elements c_i, ; If the index is itself indexed by some set, we use the notation j_i. When we refer to the j_ith cover-element we mean c_{j_i}.
- Refer to a stego key as k
- The secret message will be denoted by m, the length of m by $l(m)$, and the bits forming m by m_i, $1 \leq i \leq l(m)$. Unless otherwise stated, we assume that $m_i \in \{0, 1\}$.

2.1 Substitution Systems

A number of methods exist for hiding information in various media. These methods range from LSB coding—also known as bitplane or noise insertion tools—manipulation of image or compression algorithms to modification of image properties such as luminance. Basic substitution systems try to encode secret information by substituting insignificant parts of the cover by secret message bits; the receiver can extract the information if he has knowledge of the positions where secret information has been embedded. Since only minor modifications are made in the embedding process, the sender assumes that they will not be noticed by a passive attacker.

2.1.1 Least Significant Bit Substitution

The embedding process consists of choosing a subset {j1, . . . , jl(m)} of cover-elements and performing the substitution operation $c_{ji} \leftrightarrow$ mi on them, which exchanges the LSB of c_{ji} by mi (mi can either be 1 or 0). One could also imagine a substitution operation which changes more than one bit of the cover, for instance by storing two message bits in the two least significant bits of one cover-element. In the extraction process, the LSB of the selected cover-elements are extracted and lined up to reconstruct the secret message.

In order to be able to decode the secret message, the receiver must have access to the sequence of element indices used in the embedding process. In the simplest case, the sender uses all cover-elements for information transfer, starting at the first element. Since the secret message will normally have less bits than *l(c),* the embedding process will be finished long before the end of the cover. In this case, the sender can leave all other cover elements unchanged. This can, however, lead to a serious security problem: the first part of the cover will have different statistical properties than the second part, where no modifications have been made. To overcome this problem, more sophisticated approach is the use of a pseudorandom number generator to spread the secret message over the cover in a rather random manner; a popular approach is the *random interval method* (e.g. [1]). If both communication partners share a stego-key *k* usable as a seed for a random number generator, they can create a random sequence $k_1, \ldots, k_{l(m)}$ and use the elements with indices

$$
\begin{aligned}
j_1 &= k_1 \\
j_i &= j_{i-1} + k_i, \quad i \geq 2
\end{aligned}
$$

for information transfer. Thus, the distance between two embedded bits is determined pseudorandomly. Since the receiver has access to the seed *k* and knowledge of the pseudorandom number generator, he can reconstruct k_i and therefore the entire sequence of element indices j_i. This technique—which is especially efficient in the case of stream covers.

Learning Resource Center
Acc. No.
SP03094
JUIT, Waknaghat, Solan (H.P.)

2.1.2 Pseudorandom Permutations

If all cover bits can be accessed in the embedding process (i.e., if *c* is a random access cover), the secret message bits can be distributed randomly over the whole cover. This technique further increases the complexity for an attacker, since it is not guaranteed that subsequent message bits are embedded in the same order.

In a first attempt Alice could create (using a pseudorandom number generator) a sequence $j_1, \ldots, j_{l(m)}$ of element indices and store the *k*th message bit in the element with index j_k. Note that one index could appear more than once in the sequence, since we have not restricted the output of the pseudorandom number generator in any way. We call such a case "collision." If a collision occurs, Alice will possibly try to insert more than one message bit into one cover-element, thereby corrupting some of them.

To overcome the problem of collisions, Alice could keep track of all cover-bits which have already been used for communication in a set *B.* If during the embedding process one specific cover-element has not been used prior, she adds its index to *B* and continues to use it. If, however, the index of the cover-element is already contained in *B,* she discards the element and chooses another cover-element pseudo randomly. At the receiver side, Bob applies a similar technique.

2.1.3 Image Downgrading and Covert Channels

In 1992, Kurak and McHugh [2] reported on a security threat in high-security operating systems. Their fear was that a steganographic technique, called image downgrading, could be used to exchange images covertly. Image downgrading is a special case of a substitution system in which images act both as secret messages and covers. Given a cover-image and a secret image of equal dimensions, the sender exchanges the four least significant bits of the cover's grayscale (or color) values with the four most significant bits of the secret image. The receiver extracts the four least significant bits out of the stego-image, thereby gaining access to the most significant bits of the secret image.

While the degradation of the cover is not visually noticeable in many cases, 4 bits are sufficient to transmit a rough approximation of the secret image.

2.1.4 Cover-Regions and Parity Bits

We will call any nonempty subset of $\{c_1, \ldots, c_{l(c)}\}$ a cover-region. By dividing the cover in several disjoint regions, it is possible to store one bit of information in a whole cover-region rather than in a single element. A *parity bit* of a region I can be calculated by

$$p(I) = \sum_{j \in I} LSB(c_j) \bmod 2$$

In the embedding step, $l(m)$ disjoint cover-regions I_i $(1 \leq i \leq l(m))$ are selected, each encodes one secret bit m_i in the parity bit $p(I_i)$. If the parity bit of one cover-region I_i does not match with the secret bit m_i to encode, one LSB of the values in I_i is flipped. This will result in $p(I_i) = m_i$. In the decoding process, the parity bits of all selected regions are calculated and lined up to reconstruct the message. Again, the cover-regions can be constructed pseudo randomly using the stego-key as a seed.

2.1.5 Palette-Based Images

In a palette-based image only a subset of colors from a specific color space can be used to colorize the image. Every palette-based image format consists of two parts: a *palette* specifying N colors as a list of indexed pairs $(i, \mathbf{c}_i)$, assigning a color vector $\mathbf{c}_i$ to every index i, and the actual image data which assign a palette index to every pixel rather than the color value itself. If only a small number of color values are used throughout the image, this approach greatly reduces the file size.

Generally, there are two ways to encode information in a palette-based image: either the palette or the image data can be manipulated. The LSB of the color vectors could be used for information transfer. Alternatively, since the palette does not need to be sorted in any way, information can be encoded in the way the colors are stored in the palette. Since there are *N!* different ways to sort the palette, there is enough capacity to encode a small message. However, all methods which use the order of a palette to store information, are not robust, since an attacker can simply sort the entries in a different way and destroy the secret message (he thereby does not even modify the picture visibly).

Alternatively, information can be encoded in the image data. Since neighboring palette color values need not be perceptually similar, the approach of simply changing the LSB of some image data fails. Some steganographic applications therefore sort the palette so that neighboring colors are perceptually similar before they start the embedding process. Color values can, for instance, be stored according to their Euclidian distance in RGB space:

$$d = \sqrt{R^2 + G^2 + B^2}$$

Since the human visual system is more sensitive to changes in the luminance of a color, another (probably better) approach would be sorting the palette entries according to their luminance component. After the palette is sorted, the LSB of color indices can safely be altered.

2.1.6 Information Hiding in Binary Images

Binary images—like digitized fax data—contain redundancies in the way black and white pixels are distributed. Although the implementation of a simple substitution scheme is possible (e.g., certain pixels could be set to black or white depending on a specific message bit), these systems are highly susceptible to transmission errors and are therefore not robust.

One information hiding scheme which uses the number of black pixels in a specific image region to encode secret information was presented by Zhao and Koch [3]. A binary image is divided into rectangular image blocks B_i; let $P_0(B_i)$ be the percentage of black pixels in the image block B_i and $P_1(B_i)$ the percentage of white pixels, respectively. Basically, one block embeds a 1, if $P_1(B_i) > 50\%$ and a 0, if $P_0(B_i) > 50\%$. In the embedding process the color of some pixels is changed so that the desired relation holds. Modifications are carried out at those pixels whose neighbors have the opposite color; in sharply contrasted binary images, modifications are carried out at the boundaries of black and white pixels. These rules assure that the modifications are not generally noticeable.

2.2 Transform Domain Techniques

We have seen that LSB modification techniques are easy ways to embed information, but they are highly vulnerable to even small cover modifications. An attacker can simply apply signal processing techniques in order to destroy the secret information entirely. In many cases even the small changes resulting out of lossy compression systems yield to total information loss.

It has been noted early in the development of steganographic systems that embedding information in the frequency domain of a signal can be much more robust than embedding rules operating in the time domain. Most robust steganographic systems known today actually operate in some sort of transform domain.

Transform domain methods hide messages in significant areas of the cover image which makes them more robust to attacks, such as compression, cropping, and some image processing, than the LSB approach. However, while they are more robust to various kinds of signal processing, they remain imperceptible to the human sensory system. One method is to use the discrete cosine transformation (DCT) [4] as a vehicle to embed information in images; another would be the use of wavelet transforms [5]. Transformations can be applied over the entire image [6], to blocks throughout the image [7], or other variations. However, a trade-off exists between the amount of information added to the image and the robustness obtained [8]. Many transform domain methods are independent to image format and may survive conversion between lossless and lossy formats.

2.3 Spread Spectrum and Information Hiding

Spread spectrum (SS) communication technologies have been developed since the 1950s in an attempt to provide means of low-probability-of-intercept and antijamming communications. Pickholtz[9] define spread spectrum techniques as "means of transmission in which the signal occupies a bandwidth in excess of the minimum necessary to send the information; the band spread is accomplished by means of a code which is independent of the data, and a synchronized reception with the code at the receiver is used for despreading and subsequent data recovery." Although the power of the signal to be transmitted can be large, the signal-to-noise ratio in every frequency band will be small. Even if parts of the signal could be removed in several frequency bands, enough information should be present in the other bands to recover the signal. Thus, SS makes it difficult to detect and/or remove a signal. This situation is very similar to a steganography system which tries to spread a secret message over a cover in order to make it impossible to perceive. Since spreaded signals tend to be difficult to remove, embedding methods based on SS should provide a considerable level of robustness.

In information hiding, two special variants of SS are generally used: *direct-sequence* and *frequency-hopping* schemes. In direct-sequence schemes, the secret signal is spread by a constant called chip rate, modulated with a pseudorandom signal and added to the cover. On the other hand, in frequency-hopping schemes the frequency of the carrier signal is altered in a way that it hops rapidly from one frequency to the another.

2.4 Statistical Steganography

Statistical steganography techniques utilize the existence of "1-bit" steganographic schemes, which embed one bit of information in a digital carrier. This is done by modifying the cover in such a way that some statistical characteristics change significantly if a "1" is transmitted. Otherwise the cover is left unchanged. So the receiver must be able to distinguish unmodified covers from modified ones.

In order to construct a *l(m)*-bit stego-system from multiple "1-bit" stegosystems, a cover is divided into *l(m)* disjoint blocks $B_1, \ldots, B_{l(m)}$. A secret bit, m_i, is inserted into the ith block by placing a "1" into B_i if $m_i = 1$. Otherwise, the block is not changed in the embedding process. The detection of a specific bit is done via a test function which distinguishes modified blocks from unmodified blocks:

$$f(B_i) = \begin{cases} 1 & \text{block } B_i \text{ was modified in the embedding process} \\ 0 & \text{otherwise} \end{cases}$$

The function f can be interpreted as a hypothesis-testing function; we test the nullhypothesis "block B_i was not modified" against the alternative hypothesis "block B_i was modified." Therefore, we call the whole class of such steganography systems *statistical steganography.* The receiver successively applies f to all cover-blocks B_i in order to restore every bit of the secret message.

2.5 Distortion Techniques

In contrast to substitution systems, distortion techniques require the knowledge of the original cover in the decoding process. Alice applies a sequence of modifications to a cover in order to get a stego-object; she chooses this sequence of modifications in such a way that it corresponds to a specific secret message she wants to transmit. Bob measures the differences to the original cover in order to reconstruct the sequence of modifications applied by Alice, which corresponds to the secret message.

In many applications, such systems are not useful, since the receiver must have access to the original covers. If Wendy also has access to them, she can easily detect the cover modifications and has evidence for a secret communication. If the embedding and extraction functions are public and do not depend on a stego-key, it is also possible for Wendy to reconstruct secret messages entirely.

Distortion techniques can easily be applied to digital images. Using a similar approach as in substitution systems, the sender first chooses l(m) different cover-pixels he wants to use for information transfer. Such a selection can again be done using pseudorandom number generators or pseudorandom permutations. To encode a 0 in one pixel, the sender leaves the pixel unchanged; to encode a 1, he adds a random value Δ_x to the pixel's color. Although this approach is similar to a substitution system, there is one significant difference: the LSB of the selected color values do not necessarily equal secret message bits. In particular, no cover modifications are needed when coding a 0. Furthermore, Δ_x can be chosen in a way that better preserves the cover's statistical properties. The receiver compares all l(m) selected pixels of the stego-object with the corresponding pixels of the original cover. If the i^{th} pixel differs, the i^{th} message bit is a 1, otherwise a 0.

2.6 Cover Generation Techniques

In contrast to all embedding methods presented above, where secret information is added to a specific cover by applying an embedding algorithm, in this steganographic technique one generates a digital object only for the purpose of being a cover for secret communication.

In our project we have implemented the Least Significant Bit Substitution Method using Pseudorandom number generation for determining the maximum allowed gap between subsequent embeddings of the message.

References

[1] Möller, S., A. Pfitzmann, and I. Stirand, "Computer Based Steganography: How It Works and Why Therefore Any Restrictions on Cryptography Are Nonsense, At Best," in Information Hiding: First International Workshop, Proceedings, vol. 1174 of Lecture Notes in Computer Science, Springer, 1996, pp. 7–21.

[2] Kurak, C., and J. McHughes, "A Cautionary Note On Image Downgrading," in IEEE Computer Security Applications Conference 1992, Proceedings, IEEE Press, 1992, pp. 153–159.

[3] Zhao, J., and E. Koch, "Embedding Robust Labels into Images for Copyright Protection," in Proceedings of the International Conference on Intellectual Property Rights for Information, Knowledge and New Techniques, München, Wien: Oldenbourg Verlag, 1995, pp. 242–251.

[4] Koch, E., and J. Zhao, "Towards Robust and Hidden Image Copyright Labeling," in IEEE Workshop on Nonlinear Signal and Image Processing, Jun. 1995, pp. 452–455.

[5] Xia, X., C. G. Boncelet, and G. R. Arce, "A Multiresolution Watermark for Digital Images," in Proceedings of the IEEE International Conference on Image Processing (ICIP'97), 1997.

[6] Cox, I., et al., "A Secure, Robust Watermark for Multimedia," in Information Hiding: First International Workshop, Proceedings, vol. 1174 of Lecture Notes in Computer Science, Springer, 1996, pp. 185–206.

[7] Rhodas, G. B., "Method and Apparatus Responsive to a Code Signal Conveyed Through a Graphic Image," U.S. Patent 5,710,834, 1998.

[8] Johnson, N. F., and S. Jajodia, "Exploring Steganography: Seeing the Unseen," IEEE Computer, vol. 31, no. 2, 1998, pp. 26–34.

[9] Pickholtz, R. L., D. L. Schilling, and L. B. Milstein, "Theory of Spread-Spectrum Communications—A Tutorial," IEEE Transactions on Communications, vol. 30, no. 5, 1982, pp. 855–884.

CHAPTER 3

THE BITMAP (.bmp) FILE FORMAT

Introduction

There are many graphic file formats but mainly they are separated into two main families of graphics: raster and vector

Raster Graphics:

A raster graphics image is a data file or structure representing a generally rectangular grid of pixels, or points of color, on a computer monitor, paper, or other display device. The color of each pixel is individually defined.

Vector Graphics:

As opposed to the raster image formats above (where the data describes the characteristics of each individual pixel), vector image formats contain a geometric description which can be rendered smoothly at any desired display size.

In our project we have focused only on BMP (bit mapped) file which are *raster image* file formats.

3.1 BMP FILE

All bitmap files contain information about the image it contains. There is no "standard" format for all bitmap images, but there is a generally accepted bitmap format (primarily because of its widespread use). This is the format which follows specifications as laid down by Microsoft. Other bitmap images available are those produced by OS/2. The implementation of our project uses this widespread format only. The structure of this format is explained below.

The basic file structure

A typical bitmap file usually contains the following blocks of data and in this order:

1. **Bitmap Header**: stores general information about the bitmap file.
2. **Bitmap Information**: stores detailed information about the bitmap image.
3. **Color Palette**: stores the definition of the colors being used.
4. **Bitmap Data**: stores the actual image, pixel by pixel.

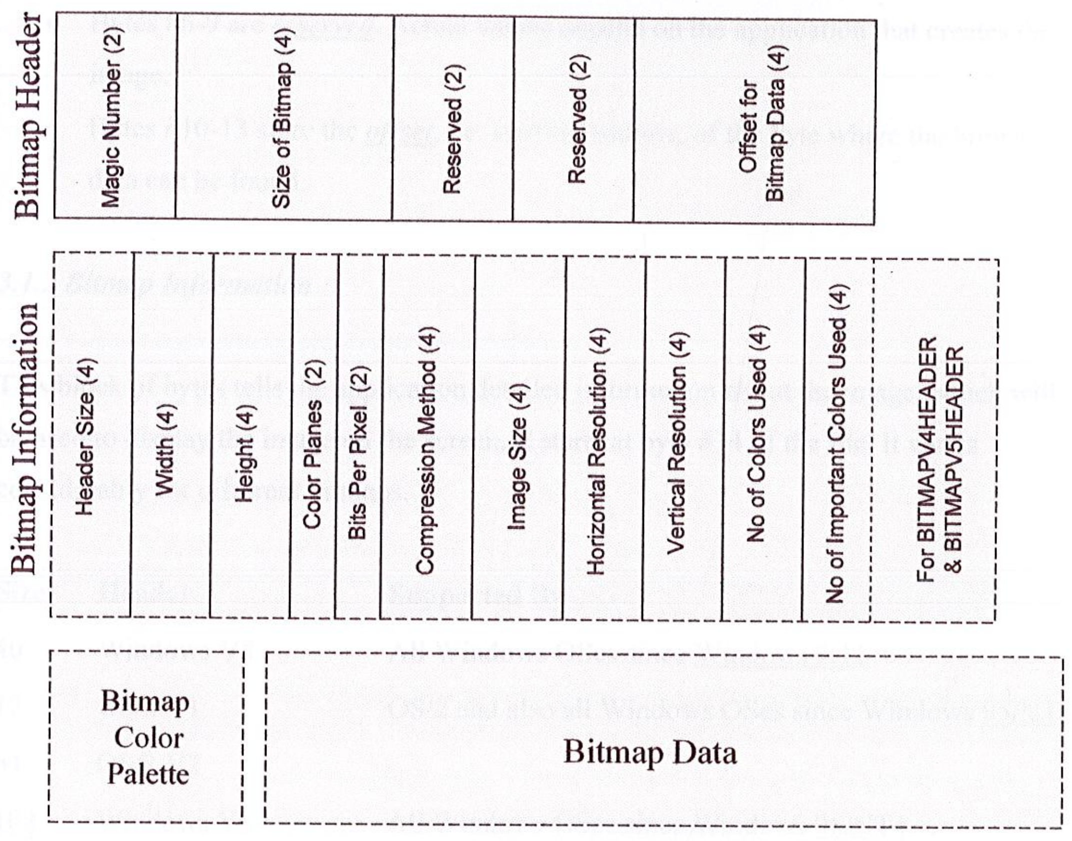


Fig.3-1 Bitmap Header & Bitmap Information

3.1.1 Bitmap Header

This block of bytes is used for identification. A typical application will read this block first to ensure that the file is actually a bitmap file and that it is not damaged. It is 14 bytes in size and is of the same structure for all bitmap variants.

- Bytes #0-1 store the ***magic number*** used to identify the bitmap file. Typical values for these 2 bytes are 0x42 0x4D (ASCII code points for 'B' and 'M'). Other variants for these two bytes may be found for OS/2 Bitmap files.
- Bytes #2-5 store the ***size*** of the bitmap file using a dword (double word).
- Bytes #6-9 are ***reserved***. Actual values depend on the application that creates the image.
- Bytes #10-13 store the ***offset***, i.e. starting address, of the byte where the bitmap data can be found.

3.1.2 Bitmap Information

This block of bytes tells the application detailed information about the image, which will be used to display the image on the screen. It starts at byte #14 of the file. It varies considerably for different bitmaps.

Size	**Header**	**Supported By**
40	Windows V3	All Windows OSes since Windows 3.11
12	OS/2 V1	OS/2 and also all Windows OSes since Windows 95/NT
64	OS/2 V2	
108	Windows V4	All Windows OSes since Windows 98/NT4
124	Windows V5	Windows 2000/XP and newer

The bytes in a Windows V3 header are arranged as:

- Bytes #14-17 specify the ***header size***. Values are: 40 – Windows V3, 12 – OS/2 V1, 64 – OS/2 V2, 108 – Windows V4, 124 – Windows V5
- Bytes #18-21 store the ***bitmap width*** in pixels.
- Bytes #22-25 store the ***bitmap height*** in pixels.
- Bytes #26-27 store the number of ***color planes*** being used. **Not often used.**
- Bytes #28-29 store the number of bits per pixel, which is the ***color depth*** of the image. Typical values are 1, 4, 8, 16, 24.

  1 – 2 Colors

  4 – 16 Colors

  8 – 256 Colors

  16 – 65536 Colors

  24 – 16.7 Million Colors
- Bytes #30-33 define the ***compression method*** being used. Possible values are 0, 1, 2, 3, 4 and 5:

  0 – none (also identified by BI_RGB)

  1 – RLE 8-bit/pixel (also identified by BI_RLE8)

  2 – RLE 4-bit/pixel (also identified by BI_RLE4)

  3 – Bit field (also identified by BI_BITFIELDS)

  4 – A JPEG image (also identified by BI_JPEG)

  5 – A PNG image (also identified by BI_PNG)

However, since most BMP images are uncompressed, the most common value is 0.

- Bytes #34-37 store the ***image size***. This is the size of the raw bitmap data (see below), and **should not be confused with the file size.**
- Bytes #38-41 store the ***horizontal resolution*** of the image. (pixel per meter)
- Bytes #42-45 store the ***vertical resolution of the image***. (pixel per meter)

- Bytes #46-49 store the ***number of colors used***. Even though $2^{\text{Color depth}}$ gives the number of total colors used, this may not represent the actual number of colors used in the color palette.
- Bytes #50-53 store the ***number of important colors used***. This number will be equal to the number of colors when every color is important. Otherwise it is used by devices that are incapable of showing those many colors mentioned in bytes # 46-39.

The above is the description of a 54-byte header found in Windows V3. Windows V4 and Windows V5 are so designed that these bytes remain the same and new information is added after these bytes. The format for an OS/2 V1 header is shown below.

- Bytes 14-17 have the value 12.
- Bytes #18-19 store the bitmap width in pixels.
- Bytes #20-21 store the bitmap height in pixels.
- Bytes #22-23 store the number of color planes being used.
- Bytes #24-25 store the number of bits per pixel.

3.1.3 Color Palette

This block of bytes defines the colors being used inside the image. As stated above, the bitmap picture will be stored pixel by pixel. Each pixel is described by a value which will be stored using one or more bytes. Therefore, the purpose of the color palette is to tell the application the actual color that each of these values corresponds to.

3.1.4 Bitmap Data

This block of bytes describes the image, pixel by pixel. Pixels are stored starting in the bottom left corner going from left to right and then row by row from the bottom to the top. Each pixel is described using one or more bytes. If the number of bytes matching a horizontal line in the image is not divisible by 4, the line is padded with null-bytes.

We have implemented a Least Bit Substitution Steganographic method for embedding message into 16-bit and 24-bit uncompressed bitmap images. This was done so as to avoid the complication involved in dealing with palette-based images. With palette based images, separate types of steganographic techniques would need to be used. Also uncompressed images were used so that Least Significant bit Substitution would be possible. Also significant distortion would be noticed using these methods with anything less than 16-bits per pixel images.

CHAPTER 4

THE PROJECT

4.1 Theory

4.1.1 Outline

The project aimed on implementing a simple steganographic method on image files using Graphical user interface for inputs and outputs. The Least Significant Bit Substitution Method was implemented with some minor modifications on 16-bit and 24-bit uncompressed bitmap images so that the embedding becomes secure and makes use a key which must be mutually shared between the participants of the communication.

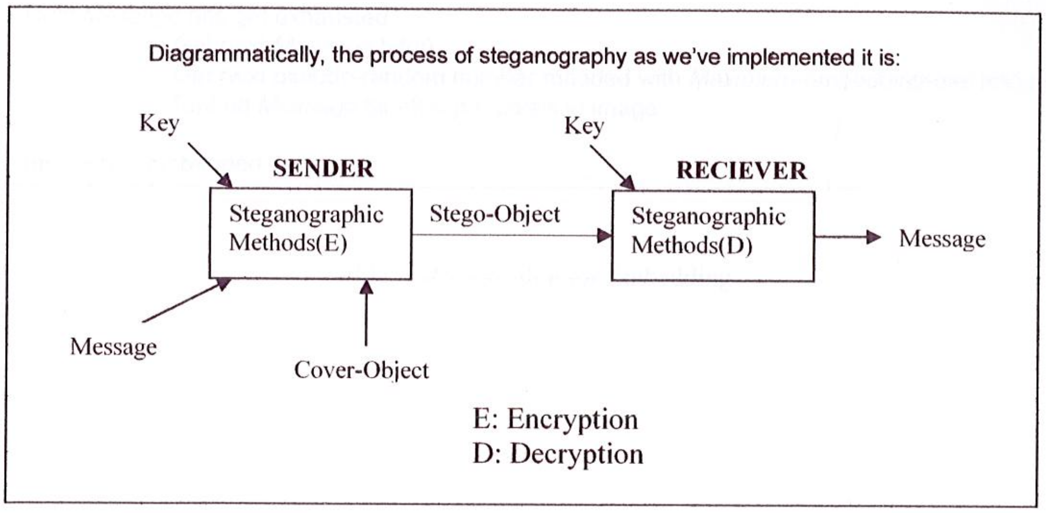

Diagrammatically, the process of steganography as we've implemented it is:
Key
SENDER
Steganographic Methods(E)
Message
Cover-Object
Stego-Object
Key
RECIEVER
Steganographic Methods(D)
Message
E: Encryption
D: Decryption

Fig. 4-1 Process of Steganography

4.1.2 Algorithms Used

Briefly, the key is used a seed to a pseudorandom number generator to generate *embedding gaps.* Message bits are embedded after these gaps in the bitmap image. This makes it a Secret-Key Steganographic technique. The algorithms are shown below.

```
Algorithm for Embedding:

Fixed-embedding-gap= constant
Maximum-embedding-gap=User-input

Initialize pseudorandom number generator with key as seed
Message = User-Input
Message-Length = Length(Message)

Until Message-Length gets exhausted
        Get next Message-Length bit
        Get next pseudorandom number modded with Fixed-embedding-gap into prn
        Embed Message-Length bit after prn pixels in image

Until Maximum-embedding-gap gets exhausted
        Get next Maximum-embedding-gap bit
        Get next pseudorandom number modded with Fixed-embedding-gap into prn
        Embed Maximum-embedding-gap bit after prn pixels in image

Until Message bits get exhausted
        Get next Message bit(s)
        Get next pseudo-random number modded with Maximum-embedding-gap into prn
        Embed Message bit after prn pixels in image

Image has embedded data
```

Fig. 4-2 Algorithm for Embedding

```
Algorithm for Extraction:

Fixed-embedding-gap= constant
Maximum-embedding-gap=?
Message-Length=?
Message=?

Initialize pseudorandom number generator with key as seed

Until Length(Message-Length) bits are extracted
        Get next pseudo-random number modded with Fixed-embedding-gap into prn
        Get next bit from image after prn pixels in image into Message-Length

Until Length(Maximum-embedding-gap) bits are extracted
        Get next pseudo-random number modded with Fixed-embedding-gap into prn
        Get next bit from image after prn pixels in image into Maximum-embedding-gap

Until Message-Length bits are extracted
        Get next pseudo-random number modded with Maximum-embedding-gap into prn
        Get next bit from image after prn pixels in image into Message

Message contains the (hidden) message to be extracted
```

Fig. 4-3 Algorithm for Extraction

Maximum-embedding-gap and Fixed-embedding-gap are variables that are used for a mod operation with the pseudorandom number that is generated. This gives us the actual number of pixels to leave before going on to the pixel into which message bits are embedded. Fixed-embedding-gap is a constant. This is needed so that the length and the Maximum-embedding-gap can be embedded into the image file to be extracted at the other end.

The Maximum-embedding-gap is made variable so that the participants of the communication can decide between more payload and more security. When the maximum-embedding-gap is more, the data will be embedded far apart, providing security in the way that the image will be distorted less as the variations will be far apart. However if the Maximum-embedding-gap is less, more message bits will be able to get into the image at the cost of the aforementioned security.

4.1.3 Embeddings in Bitmap Files

The image file used to embed data was of type : bitmap (MIME : `image/x-ms-bmp`)

Only 16-bit and 24-bit uncompressed bitmaps were used for reasons mentioned earlier at the end of section 3.1.4.

Uncompressed bitmap images were used. The structure of the pixel is shown along with the bits where message bits would be embedded.

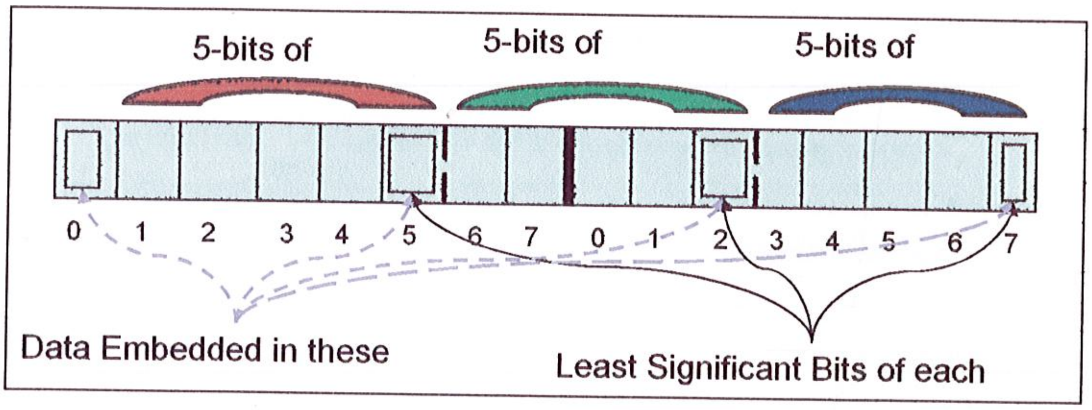


Figure 4-4 Bytes in a pixel of a 16-bit uncompressed bitmap image

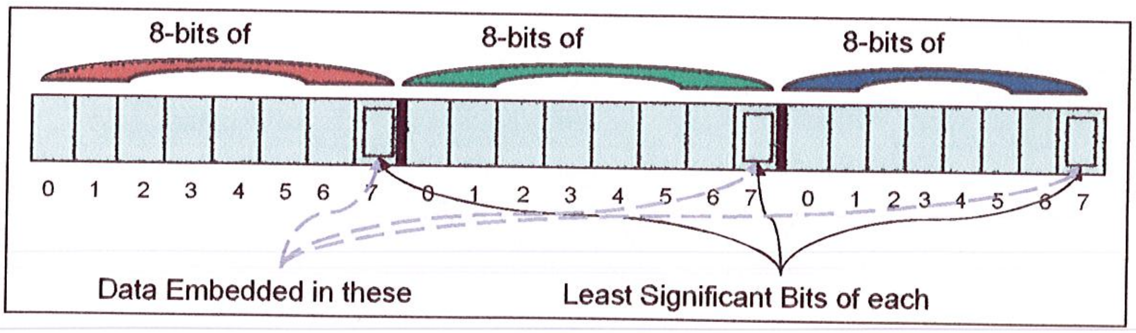


Figure 4-5 Bytes in a pixel of a 24-bit uncompressed bitmap image

The 16-bit bitmapped image has two bytes allocated to one pixel with 5-bits for each color and 1 bit unused. For the embedding process the least significant bits of these colors is used along with the 1 bit which is unused. All 4 bits are shown in the figure. Similarly for 24-bit bitmapped images, 3 bytes are allocated. Here there is no unused bit and hence in 3 bytes only 3 bits of message can be embedded.

4.2 Implementation Details

`Language Used`	C#
`Libraries Used`	.NET 2.0
`IDE`	Visual Studio 2005
`Platform`	Microsoft® Windows XP Professional
`User Interface`	Graphical (Windows Application)

The Solution is organized into 3 projects:

1. BitmapStego

   This is a class library project. It contains functions that perform steganographic embedding and extraction of message data on 16-bit and 24-bit bitmap files. It is the library that is referenced by the other two projects mentioned below.
2. LSBStegoEmbedding

   This is windows application. It contains GUI code that references the BitmapStego Project and embeds data into bitmap files.
3. LSBStegoExtraction

   This is a windows application. It contains GUI code that references BitmapStego and extracts data from the bitmap files.

The primary functions used for embedding and extracting data from 16-bit and 24-bit Bitmap files lie in a class library which is built to give a DLL file (Project: BitmapStego). This DLL file is referred to in the projects which have the GUI.

The LSBStegoEmbedding project draws three sections –

1. Input Properties
   a. Cover Object BMP : The user must browse to the cover object; After all checks are made, the disabled text field will display the path if the file is valid.
   b. Input File : The user must specify the message as a file of any extension.

    c. Key (in Hexadecimal) : The user must input a key in hexadecimal. This will be the key used in the secret-key steganography technique used to embed the data. Ideally the key should be 16 bytes or longer, though the software on its own puts no restriction on the size – upper or lower.

2. Set Payload Capacity : The payload capacity which is actually the maximum embedding gap (as shown in section 4.1.2 is specified by the user. The minimum allowed is 5 and the maximum allowed is 99.
3. Generate The Stego Object : The user must browse to the location where he/she wants to save the resulting stego-object.

The user must specify a valid cover-object, i.e. it must be a bitmap file which conforms to the Windows V3 specification (or those that are backward-compatible with it). Also it must be a 16-bit or 24-bit uncompressed bitmap image.
The program will also throw an exception and draw a dialog box informing the user if the payload-capacity settings are such that the message file cannot be embedded because its size exceeds the payload capacity of the cover-object. The user must then try and adjust the payload-settings. If this does not work, the message file simply cannot be embedded. Either a smaller message file must be tried or a larger image file must be used as the cover object.

The LSBStegoExtraction project draws 2 sections –

1. Set Input Properties
    - Stego-Object : The user must browse to the stego-object's location.
    - Key : The user must specify the key used while embedding the message. This will be the key used as secret-key steganography.
2. Extract Message File : The user must browse to the location where the message file is to be saved. The user must also know the extension of the file to be saved as this information is not embedded in the stego-object.

Exception handling has been built into the project. A friendly dialog box stating the cause of the exception is displayed.

Screenshots of the GUIs for embedding and extracting data are shown, following which flowcharts of important functions as they are implemented are shown.

4.3 Snapshots

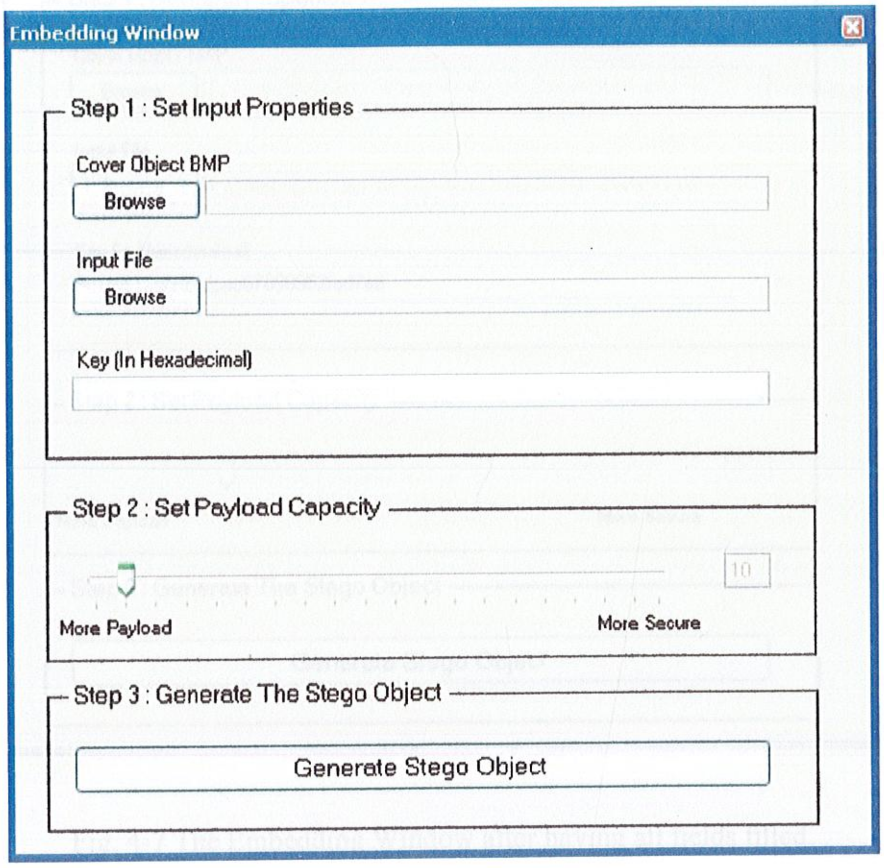
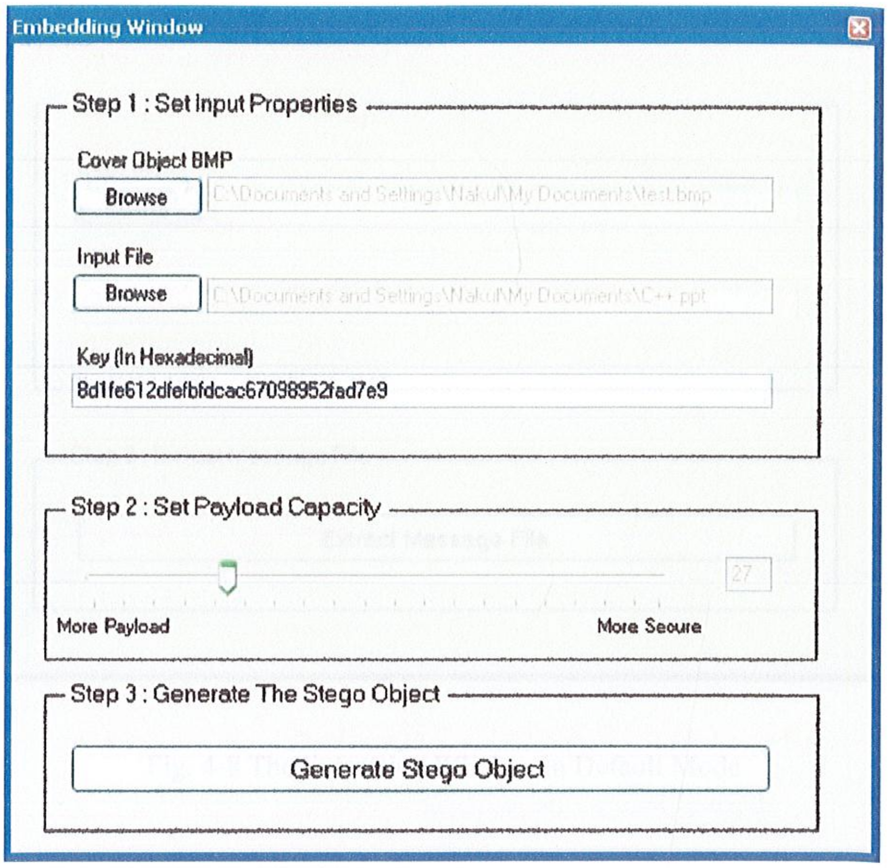


Fig. 4-6 The Embedding Window in default mode



Embedding Window

Step 1 : Set Input Properties

Cover Object BMP: C:\Documents and Settings\Nakul\My Documents\test.bmp

Input File: C:\Documents and Settings\Nakul\My Documents\C++.ppt

Key (In Hexadecimal): 8d1fe612dfefbfdcac67098952fad7e9

Step 2 : Set Payload Capacity

27

Fig. 4-7 The Embedding Window after having all fields filled.

The "Key" field contains a 128 bit hexadecimal number.

The payload capacity determines the trade-off between Payload and Security.

The user must now press the "Generate Stego Object" button and choose the destination of the file to saved.

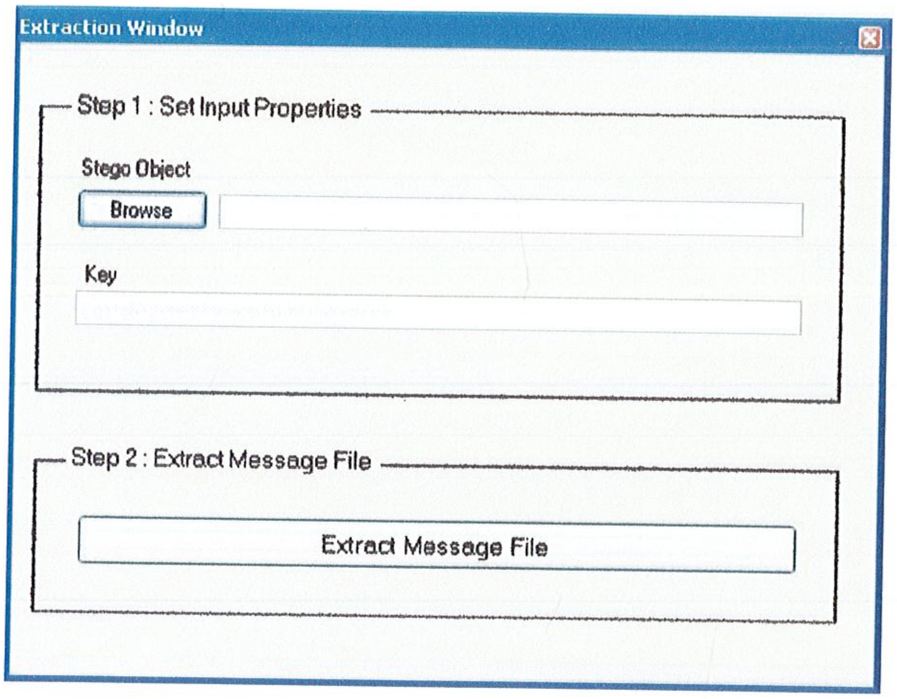
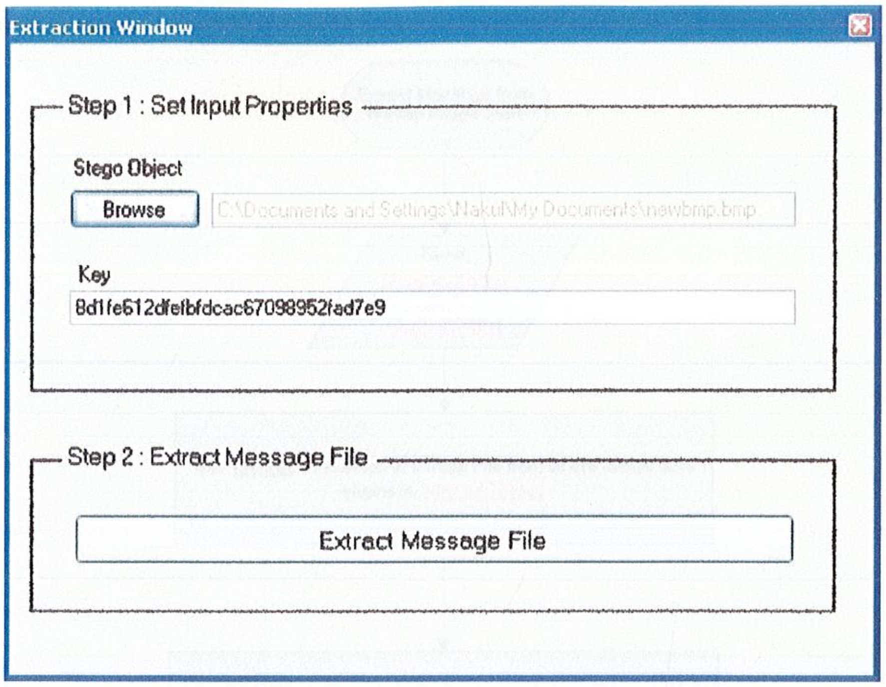


Fig. 4-8 The Extracting Window in Default Mode



Stego Object: C:\Documents and Settings\Nakul\My Documents\newbmp.bmp

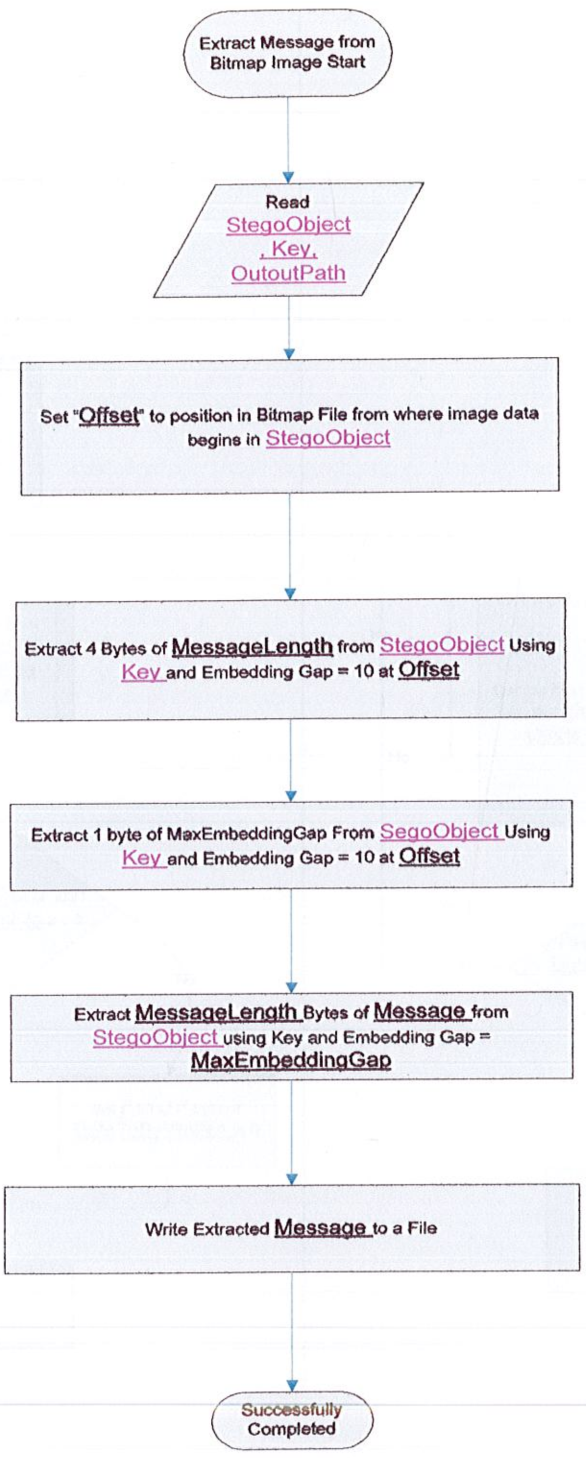
Key: 8d1fe612dfefbfdcac67098952fad7e9

Fig 4-9 The Extraction Window with all fields filled.

The "Key" is a hexadecimal number.

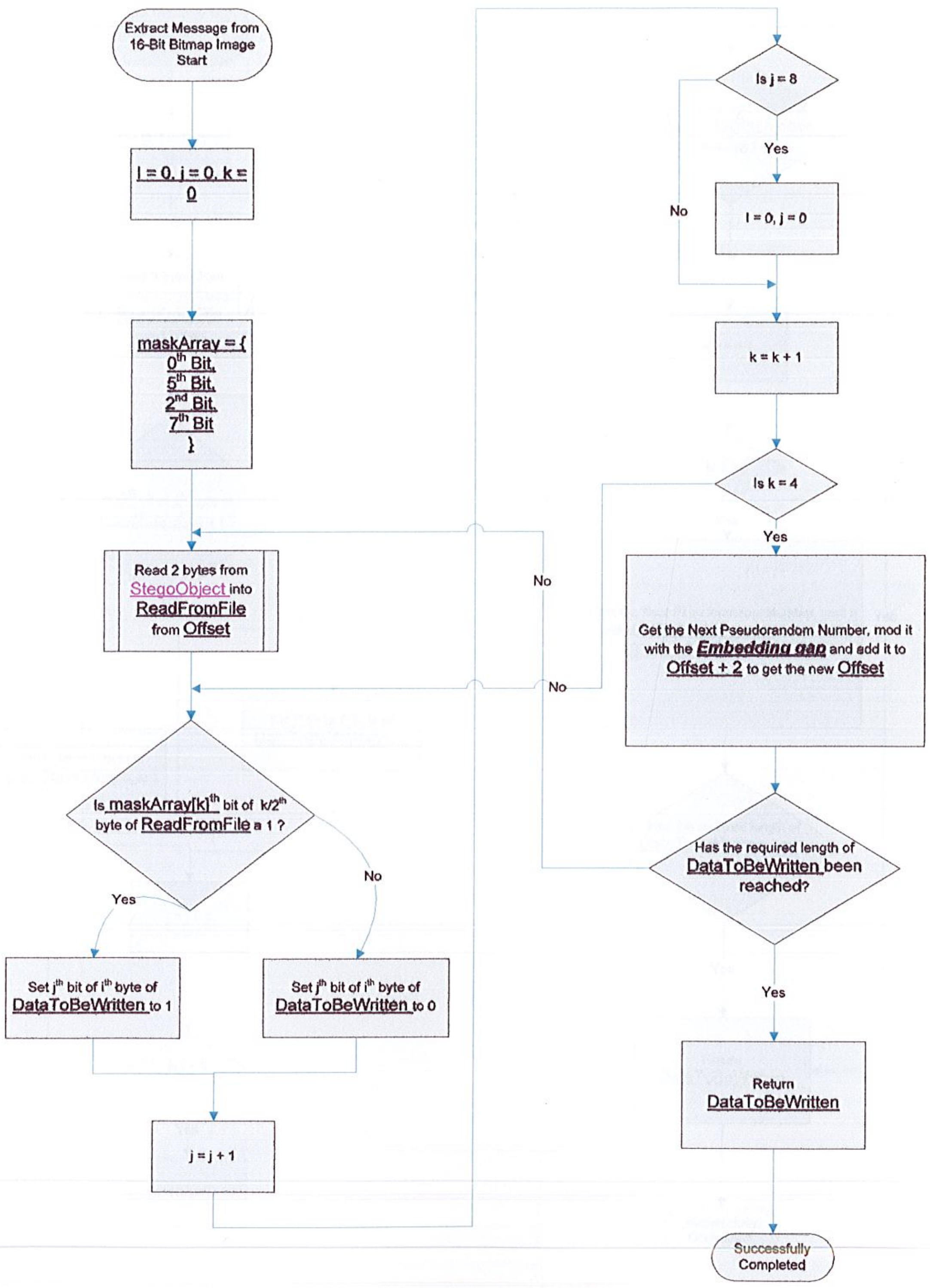
The user can now press the "Extract Message File" button and choose the location and extension of the message file to be extracted. (the receiver must know the extension of the message file).

4.4 Flowcharts



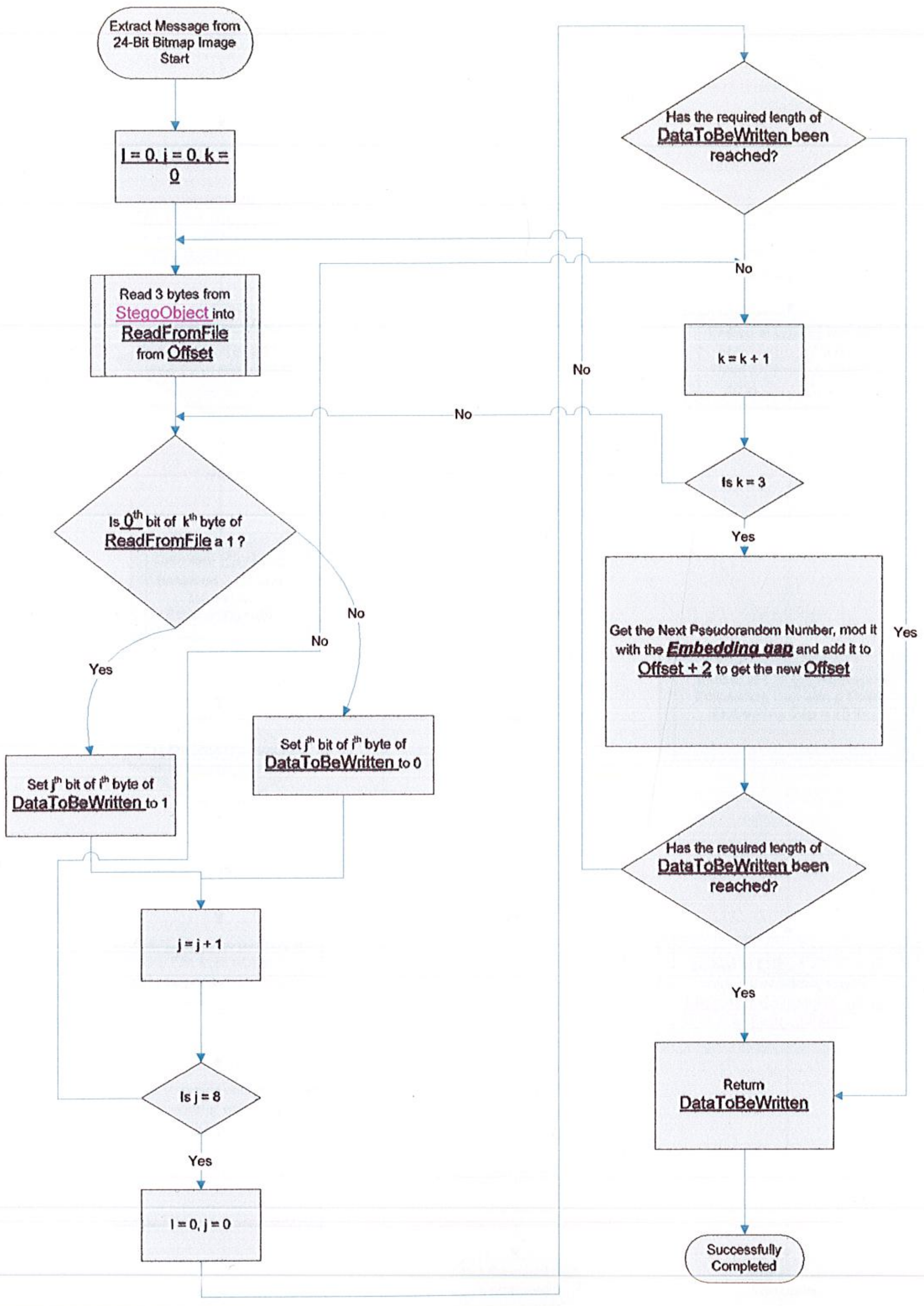
Extract Message from Bitmap Image Start
Read StegoObject , Key, OutoutPath
Set "Offset" to position in Bitmap File from where image data begins in StegoObject
Extract 4 Bytes of MessageLength from StegoObject Using Key and Embedding Gap = 10 at Offset
Extract 1 byte of MaxEmbeddingGap From SegoObject Using Key and Embedding Gap = 10 at Offset
Extract MessageLength Bytes of Message from StegoObject using Key and Embedding Gap = MaxEmbeddingGap
Write Extracted Message to a File
Successfully Completed

Fig. 4-10 Flow Chart for Extracting Message from Bitmap Image



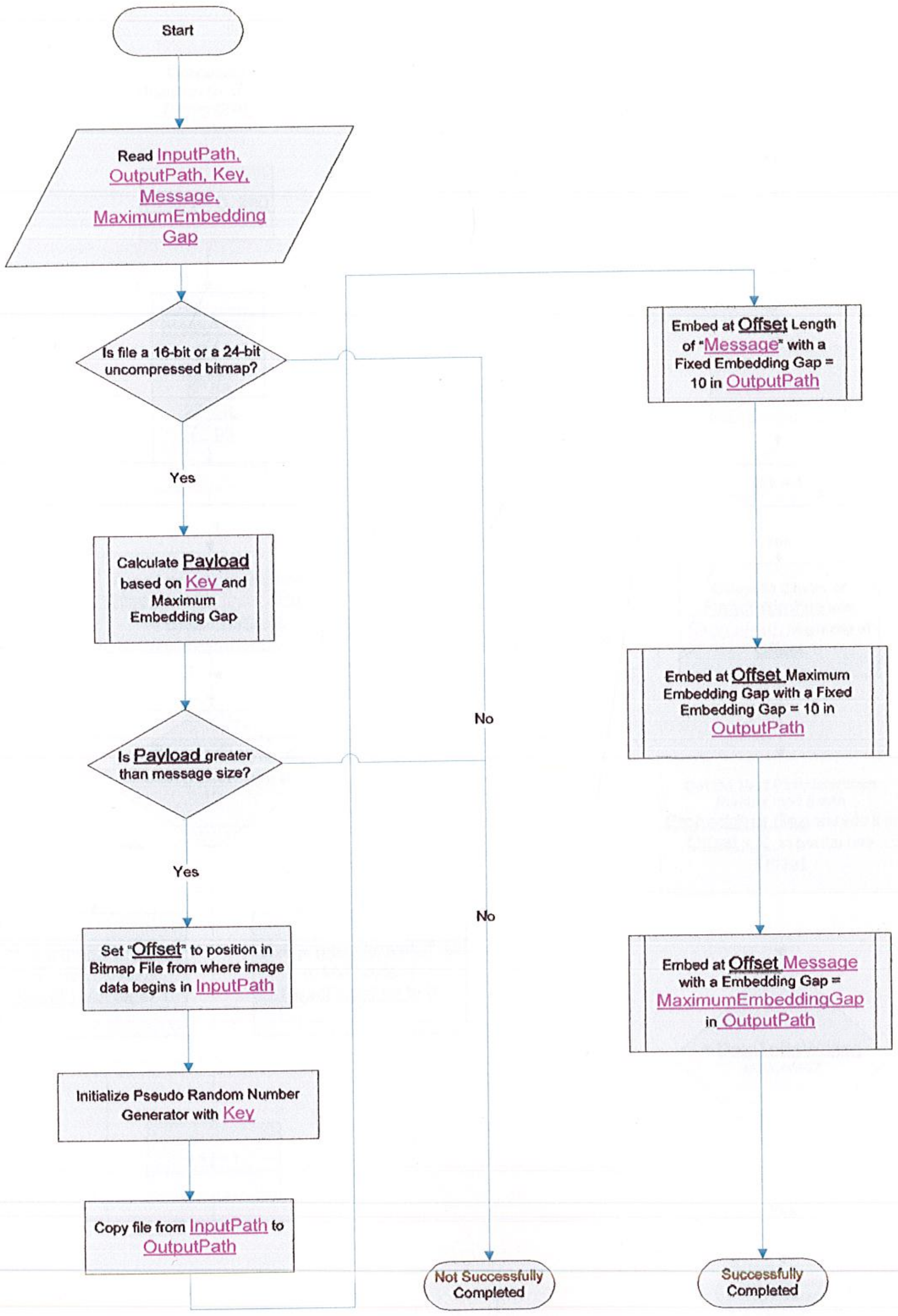
Extract Message from 16-Bit Bitmap Image Start
I = 0, j = 0, k = 0
maskArray = { 0th Bit, 5th Bit, 2nd Bit, 7th Bit }
Read 2 bytes from StegoObject into ReadFromFile from Offset
Is maskArray[k]th bit of k/2th byte of ReadFromFile a 1 ?
Yes
No
Set jth bit of ith byte of DataToBeWritten to 1
Set jth bit of ith byte of DataToBeWritten to 0
j = j + 1
Is j = 8
Yes
No
I = 0, j = 0
k = k + 1
Is k = 4
Yes
No
Get the Next Pseudorandom Number, mod it with the Embedding gap and add it to Offset + 2 to get the new Offset
Has the required length of DataToBeWritten been reached?
No
Yes
Return DataToBeWritten
Successfully Completed

Fig. 4-11 Flow Chart for Extracting Message from 16 -bit Bitmap Image



Extract Message from 24-Bit Bitmap Image Start
I = 0, j = 0, k = 0
Read 3 bytes from StegoObject into ReadFromFile from Offset
Is 0th bit of kth byte of ReadFromFile a 1 ?
Yes
No
Set jth bit of ith byte of DataToBeWritten to 1
Set jth bit of ith byte of DataToBeWritten to 0
j = j + 1
Is j = 8
Yes
I = 0, j = 0
No
Has the required length of DataToBeWritten been reached?
No
k = k + 1
Is k = 3
Yes
Get the Next Pseudorandom Number, mod it with the Embedding gap and add it to Offset + 2 to get the new Offset
Yes
Has the required length of DataToBeWritten been reached?
Yes
Return DataToBeWritten
Successfully Completed

Fig. 4-12 Flow Chart for Extracting Message from 24- bit Bitmap Image



Start
Read InputPath, OutputPath, Key, Message, MaximumEmbedding Gap
Is file a 16-bit or a 24-bit uncompressed bitmap?
Yes
No
Calculate Payload based on Key and Maximum Embedding Gap
Is Payload greater than message size?
Yes
No
Set "Offset" to position in Bitmap File from where image data begins in InputPath
Initialize Pseudo Random Number Generator with Key
Copy file from InputPath to OutputPath
Not Successfully Completed
Embed at Offset Length of "Message" with a Fixed Embedding Gap = 10 in OutputPath
Embed at Offset Maximum Embedding Gap with a Fixed Embedding Gap = 10 in OutputPath
Embed at Offset Message with a Embedding Gap = MaximumEmbeddingGap in OutputPath
Successfully Completed

Fig. 4-13 Flow Chart for Embedding Message in Bitmap Image

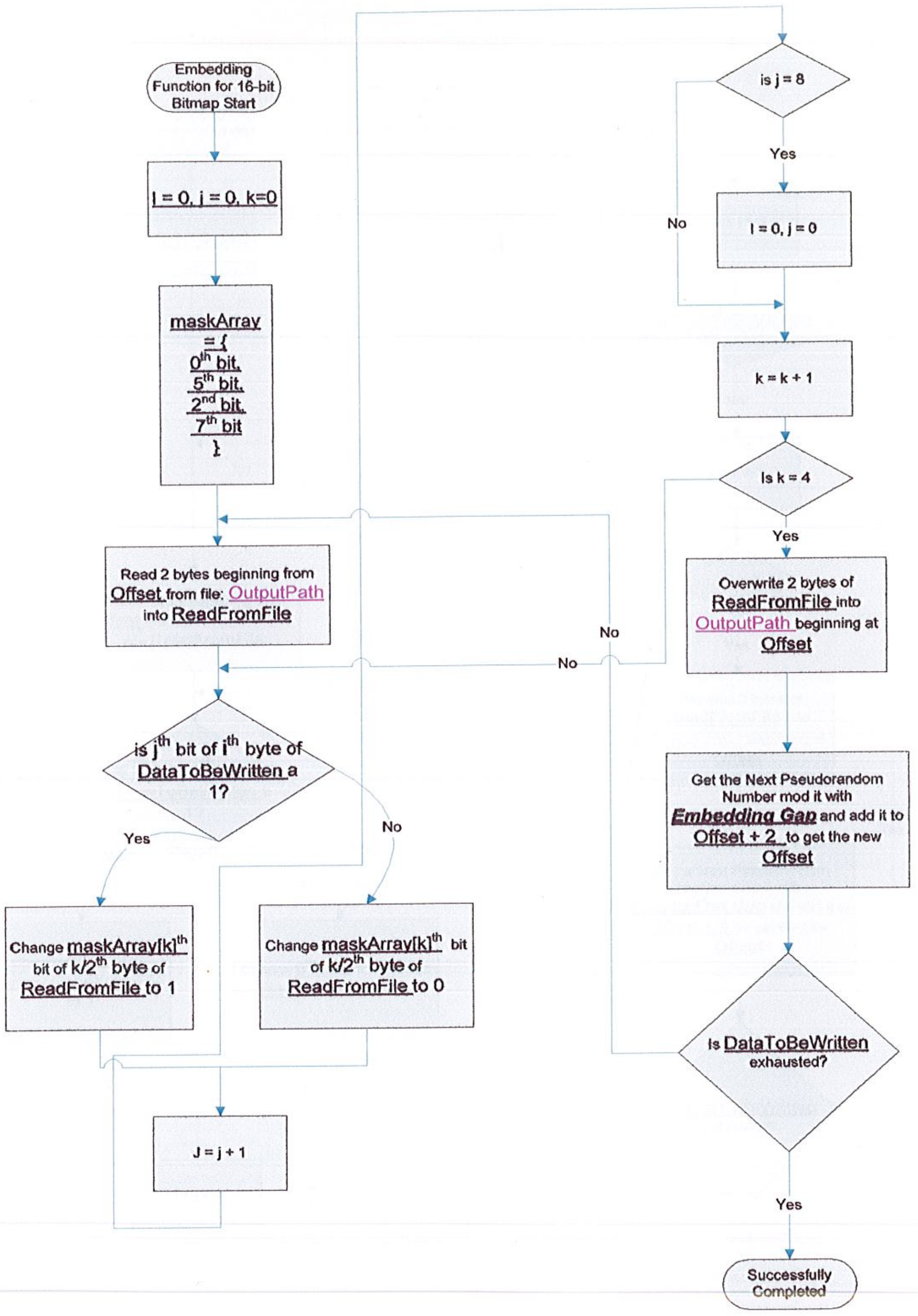
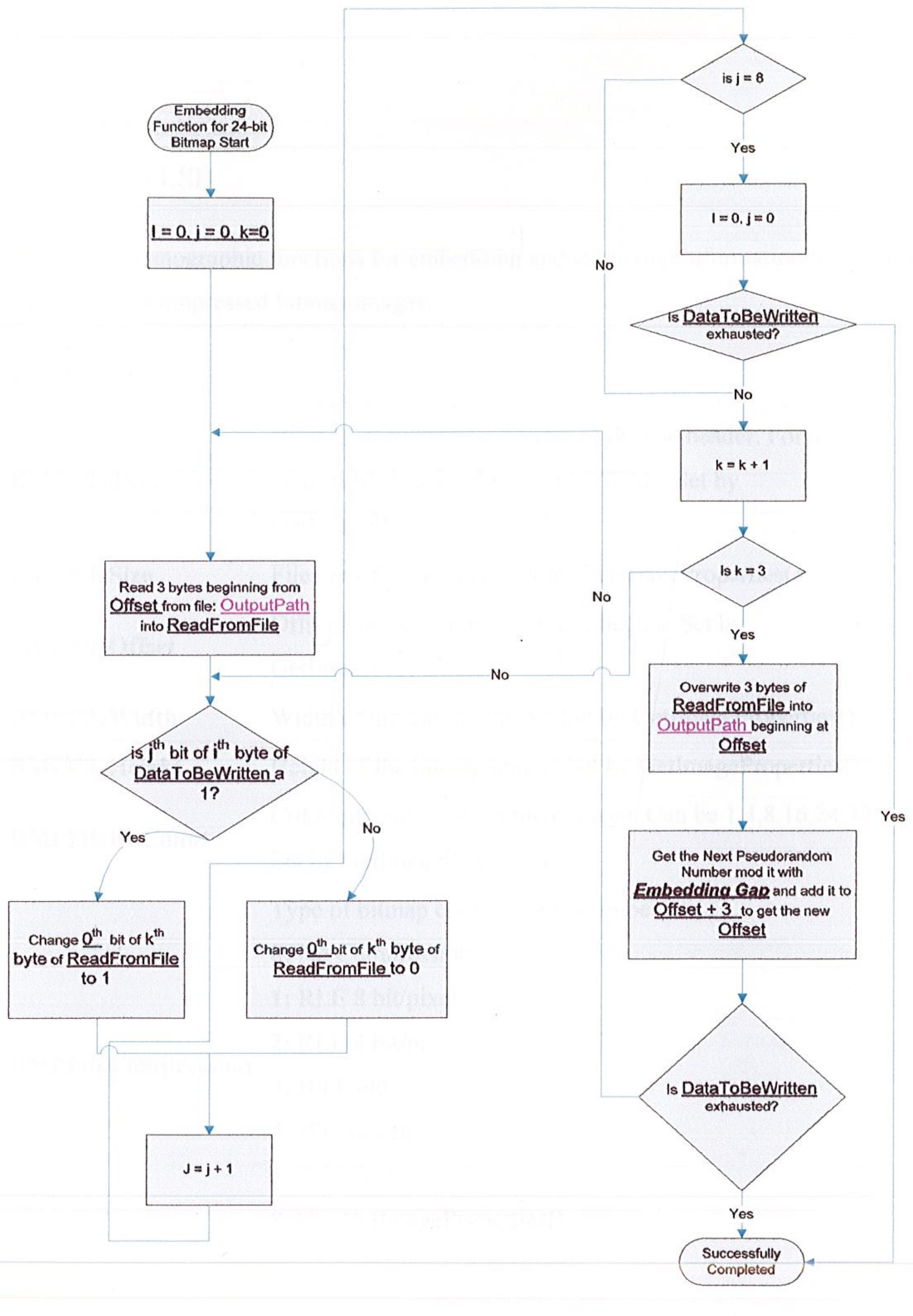

Embedding Function for 16-bit Bitmap Start
I = 0, j = 0, k=0
maskArray = { 0th bit, 5th bit, 2nd bit, 7th bit }
Read 2 bytes beginning from Offset from file: OutputPath into ReadFromFile
is jth bit of ith byte of DataToBeWritten a 1?
Yes
No
Change maskArray[k]th bit of k/2th byte of ReadFromFile to 1
Change maskArray[k]th bit of k/2th byte of ReadFromFile to 0
J = j + 1
is j = 8
Yes
No
I = 0, j = 0
k = k + 1
Is k = 4
Yes
No
No
Overwrite 2 bytes of ReadFromFile into OutputPath beginning at Offset
Get the Next Pseudorandom Number mod it with Embedding Gap and add it to Offset + 2 to get the new Offset
Is DataToBeWritten exhausted?
No
Yes
Successfully Completed

Fig. 4-14 Flow Chart for Embedding Message in 16 -bit Bitmap Image



Embedding Function for 24-bit Bitmap Start
I = 0, j = 0, k=0
Read 3 bytes beginning from Offset from file: OutputPath into ReadFromFile
is j^th bit of i^th byte of DataToBeWritten a 1?
Yes
No
Change 0^th bit of k^th byte of ReadFromFile to 1
Change 0^th bit of k^th byte of ReadFromFile to 0
J = j + 1
is j = 8
Yes
No
I = 0, j = 0
Is DataToBeWritten exhausted?
No
Yes
k = k + 1
Is k = 3
Yes
No
No
Overwrite 3 bytes of ReadFromFile into OutputPath beginning at Offset
Get the Next Pseudorandom Number mod it with Embedding Gap and add it to Offset + 3 to get the new Offset
Is DataToBeWritten exhausted?
Yes
No
Successfully Completed

Fig. 4-15 Flow Chart for Embedding Message in 24 -bit Bitmap Image

4.5 Class Documentation for Project: BitmapStego

BitmapStego (Class)

BMPSimpleLSB

Provides steganographic functions for embedding and extracting information from 16-bit and 24-bit uncompressed bitmap images.

Fields

BMPFileType	Used to store the first 2 bytes of the file header. For a compatible bitmap this must be "BM". Set by GetImageProperties()
BMPFileSize	Filesize of the bitmap. Set by GetImageProperties()
BMPFileOffset	Offset from where Bitmap data begins. Set by GetImageProperties()
BMPFileWidth	Width of the bitmap image. Set by GetImageProperties()
BMPFileHeight	Height of the bitmap image. Set by GetImageProperties()
BMPFileBitCount	Color bitcount of the bitmap image. Can be 1,4,8,16,24,32. Set by GetImageProperties()
BMPFileCompression	Type of bitmap compression. Can be **0:** No Compression **1:** RLE 8 bit/pixel **2:** RLE 4 bit/pixel **3:** Bit Field **4:** JPG Image **5:** PNG Image Set by GetImageProperties()

BMPFileColorsUsed	No of colors used in bitmap **1:** 2 **4:** 16 **8:** 256 **16:** 65536 **24:** 16777216
BMPFileImpColors	No of important colors used in bitmap
prn	Object of type Random which is seeded with the key and is used to get the next Pseudorandom number
maxEmbeddingGap	The maximum gap allowed between embeddings of data. Bigger implies lesser distortion, less payload. Smaller implied more distortion, more payload.
toRead	The *T:System.IO.FileStream* Object associated with the Original Bitmap.
toReadPath	The path of the original bitmap file
fixedEmbeddingGap	The fixed embedding gap used while embedding the Message length and Maximum Embedding Gap

Properties

MaxEmbeddingGap	To set and get the maxEmbeddingGap. This value must be between 0 and 99 **System.ArgumentOutOfRangeException:** Thrown when the maximum embedding gap is tried to be set to less than 0 or mare than 99

Methods

GetImageProperties(System.IO.FileStream)

Sets the private fields describing various properties of the bitmap image.

Parameters	**Exceptions**
fs: A *T:System.IO.FileStream*object that points to the bitmap file.	**System.IO.IOException:** Thrown when the image does not have 54 bytes to be read.

CheckImagePreRequisites(System.IO.FileStream)

Checks whether bitmap is a windows bitmap, is at least a 16-bit image and whether it is not compressed. Failing any of these conditions causes it to throw a *T:System.IO.InvalidDataException*

Parameters	**Exceptions**
fs: A *T:System.IO.FileStream*object that points to the bitmap file.	**System.IO.InvalidDataException:** Thrown when the image is either not bitmap, not 16-bit or 24-bit or is compressed

InitPRNG(System.Byte[])

Initializing the PseudoRandom Number Generator with the key

Parameters

key: The key to be used in steganography

GetNextPRN

Gets the next pseudorandomly generated number

	Return Value	
	Return Value: The next pseudorandom	

	number

CheckPayLoad(System.Byte[])

Returns payload available in bitmap file

	Return Value
Parameters **key:** The key to be used in steganography	**Return Value:** An estimate of the payload amount in bytes

Constructor: BMPSimpleLSB(System.String,System.Int32)

Constructor taking as arguements the Path of the original bitmap file and the maximum embedding gap. The image properties are checked here.

Parameters	**Exceptions**
path: The path of the Bitmap acting as the stego object **meg:** The maximum embedding gap	**System.IO.IOException:** Thrown when there is problem opening the "path" in Read-Only mode

WriteNewBMPFile(System.String,System.Byte[],System.Byte[])

Writes out the stego object - the changed bitmap file to a given location. It uses the image file passed to the constructor as the Cover Object This function calls in WriteNewBMPFile16 and WriteNewBMPFile24 to do the work depending on whether the file is a 16-bit or a 24-bit bitmap. It also creates a filestream object which it passes on to these functions.

Parameters	**Exceptions**
path: The path of the new bitmap file	**System.IO.IOException:** Thrown when there is problem in either copying the fule to "path" or there is

key: The key to be used in steganography

a problem opening the file in write mode

msg: The byte stream of the message to be embedded

WriteNewBMPFile16(System.IO.FileStream,System.Byte[],System.Byte[])

Writes out a new 16-bit bitmap file with the message embedded in it based on the key.

Parameters

fs: The *T:System.IO.FileStream* object associated with the new bitmap file

key: The key to be used in steganography

msgByteArray: The byte stream of the message to be embedded

WriteNewBMPFile24(System.IO.FileStream,System.Byte[],System.Byte[])

Writes out a new 24-bit bitmap file with the message embedded in it based on the key.

Parameters

fs: The *T:System.IO.FileStream* object associated with the new bitmap file

key: The key to be used in steganography

msgByteArray: The byte stream of the message to be embedded

ReadBMPFile(System.Byte[])

Extracts the embedded message in the stego object based on the key. The image passed to the constructor is taken as the stego object. The work is actually done by ReadBMPFile16 and ReadBMPFile24 based on whether the bitmap is 16-bits or 24-bits.

Parameters	**Return Value**	**Exceptions**
key: The key to be used in steganography	**Return Value:** A bytestream containing the message	**System.IO.IOException:** Thrown when reading from the file creates problems

ReadBMPFile16(System.IO.FileStream,System.Byte[])

Extracts the embedded message from a 16-bit bitmap image based on a key.

Parameters	**Return Value**
fs: The *T:System.IO.FileStream* object associated with the stego bitmap file **key:** A bytestream containing the message	**Return Value:** A bytestream containing the message

ReadBMPFile24(System.IO.FileStream,System.Byte[])

Extracts the embedded message from a 24-bit bitmap image based on a key.

Parameters	**Return Value**
fs: The *T:System.IO.FileStream* object	**Return Value:** A bytestream

associated with the stego bitmap file	containing the message
key: A bytestream containing the message	

4.6 Directory Structure & Files

\LSBStego	
LSBStego.sln	IDE Generated File For Solution
LSBStego.suo	IDE Generated File For Solution
\LSBStego\BitmapStego	
BitmapStego.csproj	Project File for BitmapStego Project
Class1.cs	C# Program
\LSBStego\LSBStego	
ClassDiagram1.cd	Class Diagram File
Form1.resx	Resources used by GUI
Form1.cs	Event Handling Code for GUI
Form1.Designer.cs	Code Generated by IDE
LSBStego_TemporaryKey.pfx	IDE Generated File
LSBStegoEmbedding.csproj	Project File for LSBStegoEmbedding Project
LSBStegoEmbedding.csproj.user	Project File for LSBStegoEmbedding Project
Program.cs	C# Code containing Main()
\LSBStego\LSBStegoExtraction	
Form1.resx	Resources used by GUI
Form1.cs	Event Handling Code for GUI
Form1.Designer.cs	Code Generated by IDE
LSBStegoExtraction.csproj	Project File for LSBStegoExtraction
Program.cs	C# Code containing Main()
\LSBStego\BitmapStego\obj	

BitmapStego.csproj.FileList.txt	
\LSBStego\BitmapStego\Properties	
AssemblyInfo.cs	Information About the Assembly
\LSBStego\LSBStego\Properties	
AssemblyInfo.cs	Assembly Information
Resources.resx	Resources Used by LSBStego
Resources.Designer.cs	IDE Generated File
Settings.settings	IDE Generated File
Settings.Designer.cs	IDE Generated File
\LSBStego\LSBStego\publish	
LSBStego.application	Published (Deployment) File
LSBStego_1_0_0_0.application	Published (Deployment) File
setup.exe	Published (Deployment) File
\LSBStego\LSBStegoExtraction\Properties	
AssemblyInfo.cs	Assembly Information
Resources.resx	Resources Used by LSBStego
Resources.Designer.cs	IDE Generated File
Settings.settings	IDE Generated File
Settings.Designer.cs	IDE Generated File
\LSBStego\BitmapStego\bin\Debug	
BitmapStego.XML	XML Documentation of Project BitmapStego
\LSBStego\BitmapStego\bin\Release	
BitmapStego.pdb	IDE Generated File
BitmapStego.dll	Compiled class library of project BitmapStego
\LSBStego\BitmapStego\obj\Release	
BitmapStego.pdb	IDE Generated File
BitmapStego.dll	Compiled class library of project BitmapStego
\LSBStego\LSBStego\bin\Debug	
LSBStego.XML	XML Documentation of Project LSBStegoEmbedding

LSBStego.vshost.application	Compiled GUI Application
LSBStego.vshost.exe	Compiled GUI Application
LSBStego.vshost.exe.manifest	Compiled GUI Application
LSBStego.exe	***Executable GUI Application***
\LSBStego\LSBStego\publish\LSBStego_1_0_0_0	
BitmapStego.dll.deploy	Published (Deployment) File
LSBStego.exe.deploy	Published (Deployment) File
LSBStego.exe.manifest	Published (Deployment) File
\LSBStego\LSBStegoExtraction\bin\Debug	
LSBStegoExtraction.XML	XML Documentation of Project LSBStegoExtraction
LSBStegoExtraction.vshost.exe	Compiled GUI Application
LSBStegoExtraction.exe	***Executable GUI Application***

The LSBStego.exe and LSBStegoExtraction.exe are the executable files for embedding and extraction respectively.

4.7 Compilation Environment

Architecture	X86
Processor	Intel Pentium 4 2.8GHz (HT Enabled)
Main Memory	512MB RAM
Non-Volatile Store	120GB IDE HDD
Platform	Microsoft Windows XP Professional
IDE	Microsoft Visual Studio 2005

4.8 Experimental Setup

Two experiments were conducted with two different cover objects, the first using a 24-bit bitmap image and the other a 16-bit bitmap image. An image was taken in each case as the message object. The images used as cover-object, message-object and stego-object are shown for each of the experiments. The key used and the Payload-Capacity settings are also mentioned.

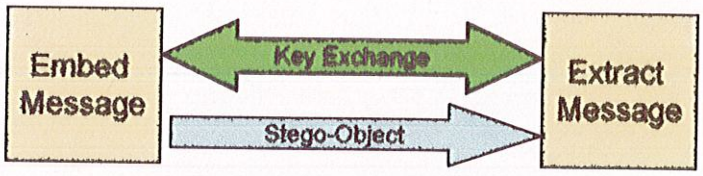


Fig. 4-16 Node Configuration

Key exchange takes place manually or through some other application.

4.8.1 Experiment 1

Embedding

Input Settings:

Fig. 4-17 Cover Object: 24-bit bitmap image (1024 X 768)

Fig. 4-18 Message Object: JPEG Image (800 X 600, 69.5 KB)

Key : ab45dbce78ede8e9d7d51dd3adacce6d

Payload – Capacity : 17

Fig. 4-19 *Output:* Stego-Object: 24-bit bitmap image (1024 X 768)

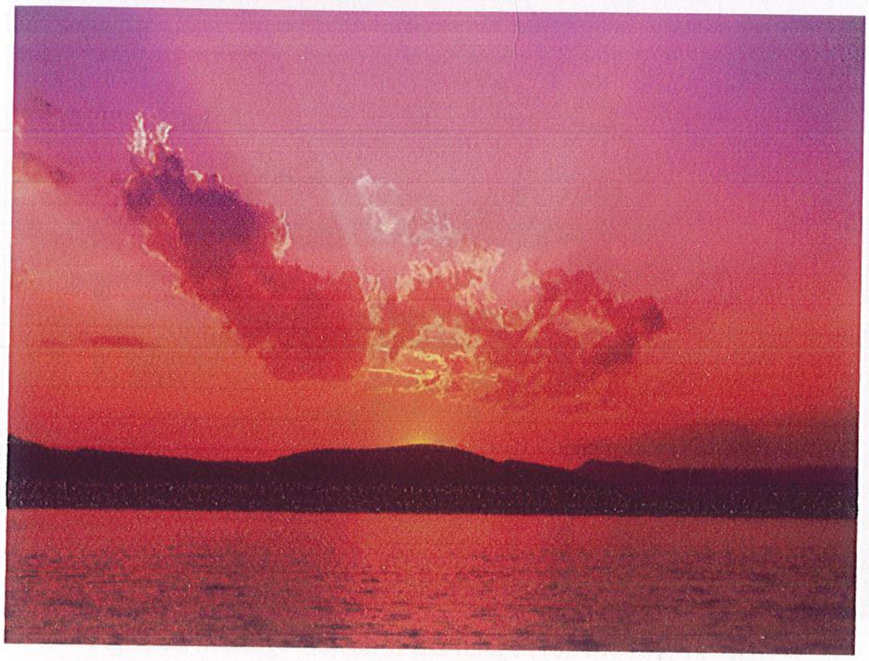
Extraction:

Input Settings

Stego-Object: As shown above.

Key: ab45dbce78ede8e9d7d51dd3adacce6d

Fig. 4-20 *Output:* Message Object on extraction:



4.8.2 Experiment 2

Embedding:

Input Settings:

Fig. 4-21 Cover Object : 16-bit bitmap image (1024 X 768):

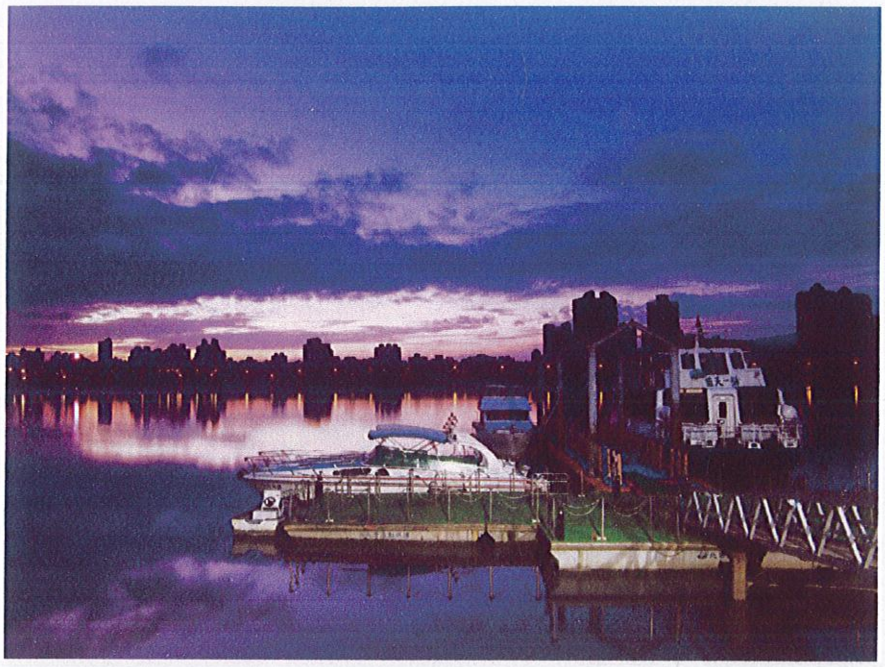
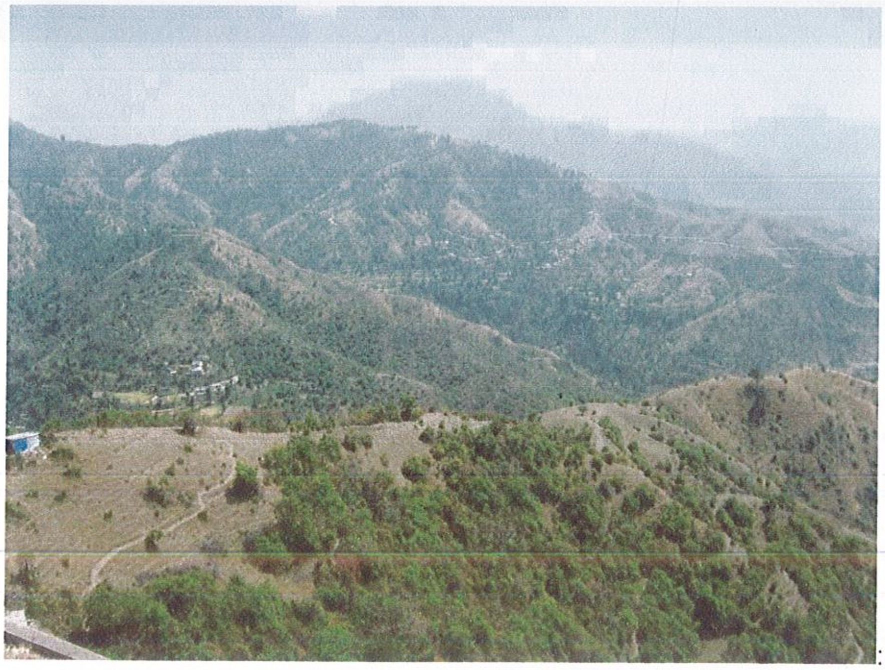


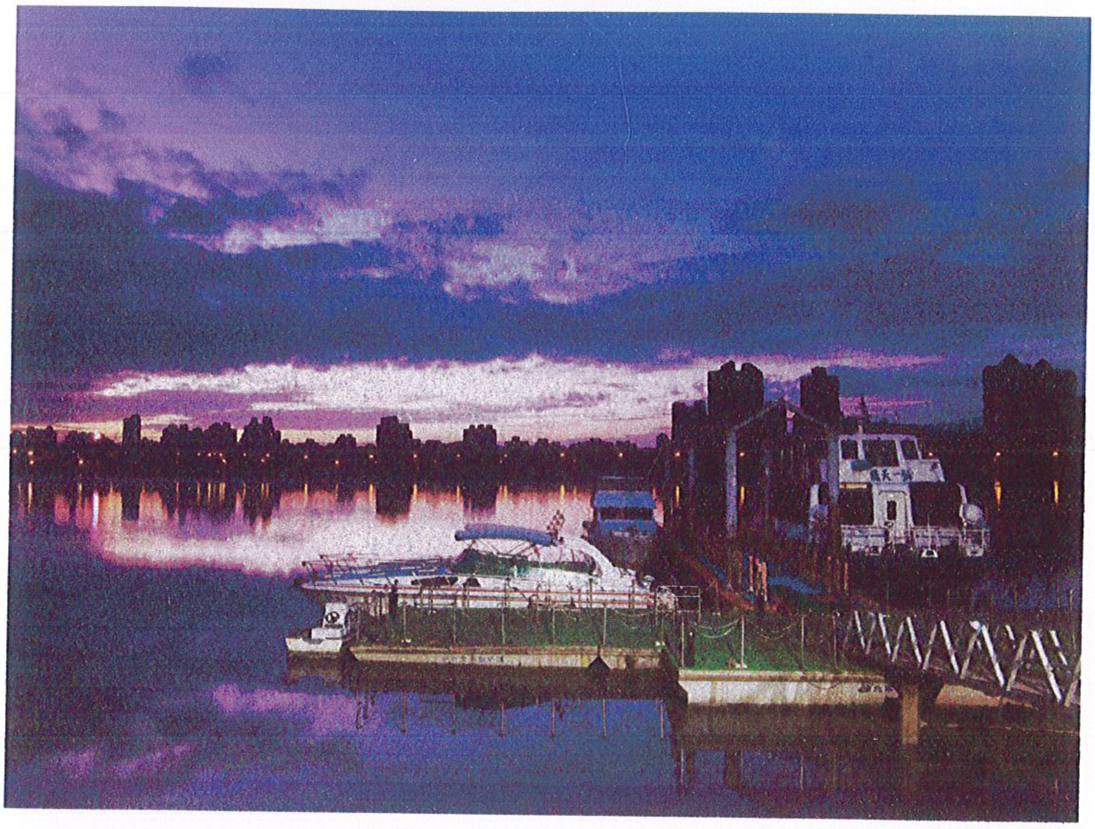
Fig. 4-22 Message Object: JPEG Image (1024 X 768, 147 KB)



Key : ab45dbce78ede8e9d7d51dd3adacce6d

Payload – Capacity : 5

Fig. 4-23 *Output:* Stego-Object: 16-bit bitmap image (1024 X 768)



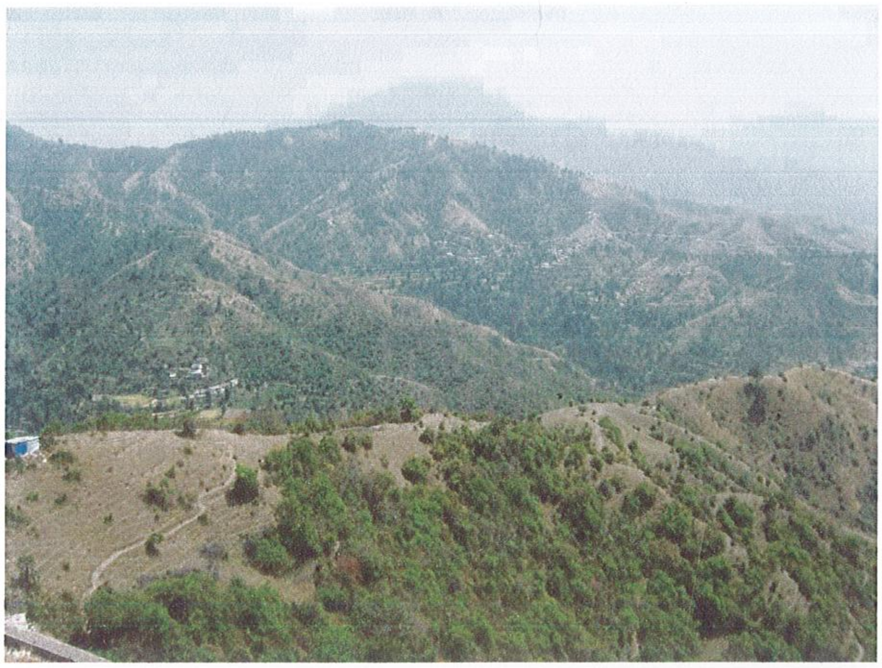
Extraction:

Input Settings:

Stego-Object: As shown above.

Key: ab45dbce78ede8e9d7d51dd3adacce6d

Fig. 4-24 *Output:* Message Object on extraction:



4.8.3 Conclusion of Experiment

There was obviously more distortion in the 16-bit image. Noise is clearly visible in the second experiment's stego-object. There was however no visible distortion in the stego-object for which a 24-bit image was taken as the cover object, in the 1st Experiment.

4.9 Code

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace BitmapStego
{

    public class
    {

        private string BMPFileType;

        private int BMPFileSize;

        private int BMPFileOffset;

        private int BMPFileWidth;

        private int BMPFileHeight;

        private int BMPFileBitCount;

        private int BMPFileCompression;
private int BMPFileColorsUsed;


private int BMPFileImpColors;




private void GetImageProperties(            fs)
{
    byte[] bmpinfobyte = new byte[54];
    fs.Read(bmpinfobyte, 0, 54);








                 utf8enc = new               ();
    BMPFileType = utf8enc.GetString(bmpinfobyte, 0, 2);
    BMPFileSize =              .ToInt32(bmpinfobyte, 2);
    BMPFileOffset =              .ToInt32(bmpinfobyte, 10);
    BMPFileWidth =              .ToInt32(bmpinfobyte, 18);
    BMPFileHeight =              .ToInt32(bmpinfobyte, 22);
    BMPFileBitCount =              .ToInt16(bmpinfobyte, 28);
    BMPFileCompression =              .ToInt32(bmpinfobyte, 30);
    BMPFileColorsUsed =              .ToInt32(bmpinfobyte, 46);
    BMPFileImpColors =              .ToInt32(bmpinfobyte, 50);
}








private void CheckImagePreRequisites(            fs)
{
    GetImageProperties(fs);
    if (BMPFileType !=      )
        throw new                   (                    );
    if (BMPFileBitCount != 16 && BMPFileBitCount != 24)
        throw new                   (
);
    if (BMPFileCompression != 0)
        throw new                   (                              );
}




        prn;
    private int maxEmbeddingGap;




    public int MaxEmbeddingGap
    {
        get
        {
            return maxEmbeddingGap;
        }
        set
        {
            if (!(value < 0 || value >= 100))
                maxEmbeddingGap = value;
            else
                throw new                                      (
                 );
        }
    }



    private void InitPRNG(byte[] key)
    {
        int initprn = 0;
        for (int i = 0; i < key.Length; i++)
        {
            initprn = (initprn + key[i]) %       .MaxValue;
        }
        prn = new       (initprn);
    }



    private int GetNextPRN()
    {
        return prn.Next();
    }



    private int CheckPayLoad(byte[] key)
    {
        InitPRNG(key);
        int totalPayload = 0;
        int toadd = 4;
        if (BMPFileBitCount == 24)
            toadd = 3;
        for (int i = BMPFileOffset; i < BMPFileSize; i += GetNextPRN() %
maxEmbeddingGap + BMPFileBitCount / 8)
        {
            totalPayload += toadd;
        }
        return totalPayload / 8;
    }



    private              toRead;
private string toReadPath;




public BMPSimpleLSB(string path, int meg)
{

    toRead = new                (path,          .Open,              .Read);
    toReadPath = path;
    MaxEmbeddingGap = meg;


    CheckImagePreRequisites(toRead);
}



private readonly int fixedEmbeddingGap = 10;






public void WriteNewBMPFile(string path, byte[] key, byte[] msg)
{
                toCreate = null;
    try
    {
        int pl = CheckPayLoad(key);
        if (msg.Length + 5 > pl)

            throw new               (
                    );


            .Copy(toReadPath, path, true);


        toCreate = new              (path,          .Open,              .ReadWrite);


        switch (BMPFileBitCount)
        {
            case 16:
                WriteNewBMPFile16(toCreate, key, msg);
                break;
            case 24:
                WriteNewBMPFile24(toCreate, key, msg);
                break;

        }
    }
    finally
    {
            if(toCreate!=null)
                toCreate.Close();
        }
    }



        private void WriteNewBMPFile16(                fs, byte[] key, byte[] msgByteArray)
        {

            InitPRNG(key);

            int toplace = BMPFileOffset;


            byte[] msgLengthArray = new byte[4];

            msgLengthArray =                    .GetBytes(msgByteArray.Length);
            byte[] tempToWrite = new byte[2];

            int i = 0,
                j = 0,
                k = 0;


            byte[] maskArray ={ 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };

            byte[] maskArray16 ={ 0x80, 0x04, 0x20, 0x01 };

            byte[] maskArrayComp16 ={ 0x7F, 0xFB, 0xDF, 0xFE };

            byte tmp;

            while (msgLengthArray.Length != i)
            {

                fs.Seek(toplace,            .Begin);
                fs.Read(tempToWrite, 0, 2);


                for (k = 0; k < 4; k++)
                {
                    tmp = (byte)(msgLengthArray[i] & maskArray[j]);
                    if (tmp != 0x00)
                    {
                        tempToWrite[(int)k / 2] = (byte)(tempToWrite[(int)k / 2] |
maskArray16[k]);
                    }
                    else
                    {
                        tempToWrite[(int)k / 2] = (byte)(tempToWrite[(int)k / 2] &
maskArrayComp16[k]);
                    }

                    j = (j + 1) % 8;
                    if (j == 0)
                    {
                        i++;
                    }
                }
            //write back the [illegible]
            fs.Seek(toplace,             .Begin);
            fs.Write(tempToWrite, 0, 2);

            [illegible]
            toplace += GetNextPRN() % fixedEmbeddingGap + 2;
        }

        [illegible]

        byte meg = (byte)maxEmbeddingGap;
        i = 0;
        while (i!=1)
        {
            fs.Seek(toplace,             .Begin);
            fs.Read(tempToWrite, 0, 2);

            for (k = 0; k < 4; k++)
            {
                tmp = (byte)(meg & maskArray[j]);
                if (tmp != 0x00)
                {
                    tempToWrite[(int)k / 2] = (byte)(tempToWrite[(int)k / 2] |      ↙
maskArray16[k]);
                }
                else
                {
                    tempToWrite[(int)k / 2] = (byte)(tempToWrite[(int)k / 2] &      ↙
maskArrayComp16[k]);
                }

                j = (j + 1) % 8;
                if (j == 0)
                {
                    i++;
                }
            }
            fs.Seek(toplace,             .Begin);
            fs.Write(tempToWrite, 0, 2);
            toplace += GetNextPRN() % fixedEmbeddingGap + 2;
        }



        i = 0;
        while (msgByteArray.Length != i)
        {
            fs.Seek(toplace,             .Begin);
            fs.Read(tempToWrite, 0, 2);

            for (k = 0; k < 4; k++)
            {
                tmp = (byte)(msgByteArray[i] & maskArray[j]);
                if (tmp != 0x00)
                {
                    tempToWrite[(int)k / 2] = (byte)(tempToWrite[(int)k / 2] |      ↙
maskArray16[k]);
                }
                else
                {
                    tempToWrite[(int)k / 2] = (byte)(tempToWrite[(int)k / 2] &      ↙
maskArrayComp16[k]);
                }

                j = (j + 1) % 8;
                if (j == 0)
                {
                    i++;
                }
            }
            fs.Seek(toplace,           .Begin);
            fs.Write(tempToWrite, 0, 2);
            toplace += GetNextPRN() % maxEmbeddingGap + 2;
        }
    }




    [illegible] WriteNewBMPFile24(           fs, byte[] key, byte[] msgByteArray)
    {


        InitPRNG(key);
        int toplace = BMPFileOffset;

        byte[] msgLengthArray = new byte[4];
        msgLengthArray =               .GetBytes(msgByteArray.Length);
        byte[] tempToWrite = new byte[3];

        int i = 0,
            j = 0,
            k = 0;


        byte[] maskArray ={ 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };

        byte tmp;
        while (msgLengthArray.Length != i)
        {
            fs.Seek(toplace,           .Begin);
            fs.Read(tempToWrite, 0, 3);

            for (k = 0; k < 3; k++)
            {
                tmp = (byte)(msgLengthArray[i] & maskArray[j]);
                if (tmp != 0x00)
                {
                    tempToWrite[k] = (byte)(tempToWrite[k] | 0x01);
                }
                else
                {
                    tempToWrite[k] = (byte)(tempToWrite[k] & 0xFE);
                }

                j = (j + 1) % 8;
                if (j == 0)
                {
                    i++;
                    if (i == msgLengthArray.Length)
                        break;
                }
            }
            fs.Seek(toplace,           .Begin);
            fs.Write(tempToWrite, 0, 3);
            toplace += GetNextPRN() % fixedEmbeddingGap + 3;
        }

        byte meg = (byte)maxEmbeddingGap;
        i = 0;
        while (1!= i)
        {
            fs.Seek(toplace,           .Begin);
            fs.Read(tempToWrite, 0, 3);

            for (k = 0; k < 3; k++)
            {
```

```
            tmp = (byte)(meg & maskArray[j]);
            if (tmp != 0x00)
            {
                tempToWrite[k] = (byte)(tempToWrite[k] | 0x01);
            }
            else
            {
                tempToWrite[k] = (byte)(tempToWrite[k] & 0xFE);
            }

            j = (j + 1) % 8;
            if (j == 0)
            {
                i++;
                if (1 == i)
                    break;
            }
        }
        fs.Seek(toplace,           .Begin);
        fs.Write(tempToWrite, 0, 3);
        toplace += GetNextPRN() % fixedEmbeddingGap + 3;
    }


    i = 0;
    while (msgByteArray.Length != i)
    {
        fs.Seek(toplace,           .Begin);
        fs.Read(tempToWrite, 0, 3);

        for (k = 0; k < 3; k++)
        {
            tmp = (byte)(msgByteArray[i] & maskArray[j]);
            if (tmp != 0x00)
            {
                tempToWrite[k] = (byte)(tempToWrite[k] | 0x01);
            }
            else
            {
                tempToWrite[k] = (byte)(tempToWrite[k] & 0xFE);
            }

            j = (j + 1) % 8;
            if (j == 0)
            {
                i++;
                if (msgByteArray.Length == i)
                    break;
            }
        }
        fs.Seek(toplace,           .Begin);
        fs.Write(tempToWrite, 0, 3);
        toplace += GetNextPRN() % maxEmbeddingGap + 3;
    }
}




public byte[] ReadBMPFile(byte [] key)
{


              toExtract = null;
        byte[] toreturn = null;
        try
        {

            toExtract = toRead;


            switch (BMPFileBitCount)
            {
                case 16:
                    toreturn = ReadBMPFile16(toExtract, key);
                    break;
                case 24:
                    toreturn = ReadBMPFile24(toExtract, key);
                    break;
            }
        }
        finally
        {

            toExtract.Close();
        }
        return toreturn;
    }






    private byte[] ReadBMPFile16(              fs, byte[] key)
    {
        int i = 0,
            j = 0,

            k = 0;


        byte[] msgLengthArray = new byte[4];
        int toplace = BMPFileOffset;


        InitPRNG(key);


        byte[] tempToRead = new byte[2];



        byte[] maskArray ={ 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };
        byte[] maskArray16 ={ 0x80, 0x04, 0x20, 0x01 };
        byte[] maskArrayComp16 ={ 0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F };

        byte tmp;


        while (msgLengthArray.Length != i)
        {

            fs.Seek(toplace,            .Begin);
            fs.Read(tempToRead, 0, 2);


            for (k = 0; k < 4; k++)
            {
                tmp = (byte)(tempToRead[(int)k / 2] & maskArray16[k]);
                if (tmp != 0x00)
                {
                    msgLengthArray[i] = (byte)(msgLengthArray[i] | maskArray[j]);
                }
                else                    //Embed a 0
                {
                    msgLengthArray[i] = (byte)(msgLengthArray[i] & maskArrayComp16 ↙
[j]);
                }

                j = (j + 1) % 8;
                if (j == 0)
                {
                    i++;
                }
            }
            //Get the next Pseudorandom number
            toplace += GetNextPRN() % fixedEmbeddingGap + 2;
        }

        /************* Extract the Maximum Embedding Gap ***************/        ↙

        //Functionally similar to the code given above
        i = 0;
        byte meg=(byte)maxEmbeddingGap;
        while (1 != i)
        {
            fs.Seek(toplace, [illegible].Begin);
            fs.Read(tempToRead, 0, 2);

            for (k = 0; k < 4; k++)
            {
                tmp = (byte)(tempToRead[(int)k / 2] & maskArray16[k]);
                if (tmp != 0x00)
                {
                    meg = (byte)(meg | maskArray[j]);
                }
                else
                {
                    meg = (byte)(meg & maskArrayComp16[j]);
                }

                j = (j + 1) % 8;
                if (j == 0)
                {
                    i++;
                }
            }
            toplace += GetNextPRN() % fixedEmbeddingGap + 2;
        }
        //Set the Maximum Embedding Gap of this object to the value extracted from ↙
the file
        maxEmbeddingGap = meg;

        /********** Extract the message stream *************/
        //Functionally similar to the code above

        i = 0;
        int msglength = [illegible].ToInt32(msgLengthArray, 0);
        byte[] msgArray = new byte[msglength];

        while (msgArray.Length != i)
        {
            fs.Seek(toplace, [illegible].Begin);
            fs.Read(tempToRead, 0, 2);

            for (k = 0; k < 4; k++)
            {
                tmp = (byte)(tempToRead[(int)k / 2] & maskArray16[k]);
                if (tmp != 0x00)
                {
                    msgArray[i] = (byte)(msgArray[i] | maskArray[j]);
                }
                else
                {
                    msgArray[i] = (byte)(msgArray[i] & maskArrayComp16[j]);
                }

                j = (j + 1) % 8;
                if (j == 0)
                {
                    i++;
                }
            }
            toplace += GetNextPRN() % maxEmbeddingGap + 2;
        }
        return msgArray;
    }



    public static byte[] ReadBMPFile24(            fs, byte[] key)
    {



        int i = 0, j = 0, k = 0;
        byte[] msgLengthArray = new byte[4];
        int toplace = BMPFileOffset;
        InitPRNG(key);
        byte[] tempToRead = new byte[3];
        byte[] maskArray ={ 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };
        byte[] maskArrayComp16 ={ 0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F };
        byte tmp;
        while (msgLengthArray.Length != i)
        {

            fs.Seek(toplace,           .Begin);
            fs.Read(tempToRead, 0, 3);


            for (k = 0; k < 3; k++)
            {
                tmp = (byte)(tempToRead[k] & 0x01);
                if (tmp != 0x00)
                {
                    msgLengthArray[i] = (byte)(msgLengthArray[i] | maskArray[j]);
                }
                else
                {
                    msgLengthArray[i] = (byte)(msgLengthArray[i] & maskArrayComp16 ↙
[j]);
                }

                j = (j + 1) % 8;
                if (j == 0)
                {
                    i++;
                    if (i == msgLengthArray.Length)
                        break;
                }
            }
            toplace += GetNextPRN() % fixedEmbeddingGap + 3;
        }

        i = 0;

        byte meg=(byte)maxEmbeddingGap;

        while (1!= i)
        {
            fs.Seek(toplace,           .Begin);
            fs.Read(tempToRead, 0, 3);
```

```
                for (k = 0; k < 3; k++)
                {
                    tmp = (byte)(tempToRead[k] & 0x01);
                    if (tmp != 0x00)
                    {
                        meg = (byte)(meg | maskArray[j]);
                    }
                    else
                    {
                        meg = (byte)(meg & maskArrayComp16[j]);
                    }

                    j = (j + 1) % 8;
                    if (j == 0)
                    {
                        i++;
                        if (i == 1)
                            break;
                    }
                }
                toplace += GetNextPRN() % fixedEmbeddingGap + 3;
            }
            maxEmbeddingGap = meg;


            i = 0;
            int msglength =            .ToInt32(msgLengthArray, 0);
            byte[] msgArray = new byte[msglength];

            while (msgArray.Length != i)
            {
                fs.Seek(toplace,           .Begin);
                fs.Read(tempToRead, 0, 3);

                for (k = 0; k < 3; k++)
                {
                    tmp = (byte)(tempToRead[k] & 0x01);
                    if (tmp != 0x00)
                    {
                        msgArray[i] = (byte)(msgArray[i] | maskArray[j]);
                    }
                    else
                    {
                        msgArray[i] = (byte)(msgArray[i] & maskArrayComp16[j]);
                    }

                    j = (j + 1) % 8;
                    if (j == 0)
                    {
                        i++;
                        if (i == msgArray.Length)
                            break;
                    }
                }
                toplace += GetNextPRN() % maxEmbeddingGap + 3;
            }
            return msgArray;
        }
    }
}
```

Conclusion

A study of the various steganographic methods for embedding data into image files was studied. The Least Significant Bit Substitution method with some modifications was implemented. After conducting experiments, we concluded that 16-bit bitmap images get significantly distorted when data is embedded into them. However for 24-bit bitmap images, the distortions are not visible.

Additionally, the project can be extended by implementing LSB techniques for other uncompressed image file format or those with color palette data in them. Also more robust methods like embedding data in another transform domain could be implemented.

An auxiliary utility to exchange the secret keys can be implemented.

Bibliography

Study of Steganography & Image File Formats

- Stefan Katzenbeisser, Fabien A.P. Petitcolas *Information Hiding Techniques for Steganography and Digital Watermarking* Artech House Inc, 2000.
- Chun-Shien Lu *Multimedia Security : Steganography and Digital Watermarking Techniques for Protection Of Intellectual* Property Idea Group Publishing, 2004
- Alvaro Martín, Guillermo Sapiro, and Gadiel Seroussi,'' Is Image Steganography Natural?'', Fellow, IEEE Transaction on Image Processing, Vol. 14,No.12, December 2005
- Giuseppe Mastronardi, Marcello Castellano, Francescomaria Marino," Steganography Effects in Various Formats of Images. A Preliminary Study , International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications 1-4 July 2001. Foros. Ukraine , 0-7803-7164-IXO 1 I 2001 IEEE
- Samer Atawneh, ''A New Algorithm for Hiding Gray Images Using Blocks '', 0-7803-9521-2/06/§2006 IEEE.
- Lisa M. Marvel and Charles T. Retter, Charles G. Boncelet, Jr.," Hiding Information in Images", 0-8186-8821-1198 0 1998 IEEE
- Mehdi Kharrazi, Husrev T. Sencar, and Nasir Memon, "Image Steganography: Concepts and Practice", April 22, 2004 1:49 WSPC/Lecture Notes Series: 9in x 6in
- Lisa M. Marvel and Charles T. Retter, Charles G. Boncelet, Jr., "A Methodology For Data Hiding Using Images", 0-7803-4506-1/98/ 01 998 IEEE.
- Hide and Seek- An Introduction to Steganography- Security & Privacy, Niels Provos And Peter Honeyman, Magazine-IEEE
- Digital Steganography: Hiding Data within Data, Donovan Artz • Los Alamos National Laboratory,IEEE
- H.-C. Wu, N.-I. Wu, C.-S. Tsai and M.-S. Hwang, " Image steganographic scheme based on pixel-value differencing and LSB replacement methods '', IEEE Proc.-Vis. Image Signal Process., Vol. 152, No. 5, October 2005

- Chi-Shiang Chan and Chin-Chen Chang, "An Image Hiding Scheme Based on Multi-bit-reference Substitution Table Using Dynamic Programming Strategy ", Fundamenta Informaticae 65 (2005) 291–305, IOS Press
- Kevin Curran, "An Evaluation of Image Based Steganography Methods ", International Journal of Digital Evidence Fall 2003, Volume 2, Issue 2
- Jessica Fridrich SUNY Binghamton, "Minimizing the Embedding Impact in Steganography", MM&Sec'06, September 26–27, 2006, Geneva, Switzerland. Copyright 2006 ACM 1-59593-493-6/06/0009
- R.Chandramouli,Nasir Memon, " Analysis of LSB Based Image Steganographic Techniques ",0-7803-6725-1/01/ IEEE
- Bret Dunbar, "A detailed look at Steganographic Techniques and their use in an Open-Systems Environment", © SANS Institute 2002, As part of the Information Security Reading Room.
- Ran-Zan Wang and Yeh-Shun Chen, "High-Payload Image Steganography Using Two-Way Block Matching ", Manuscript received August 17, 2005; revised October 19, 2005.IEEE
- Armando J. Pinho and António J. R. Neves, "A Survey on Palette Reordering Methods for Improving the Compression of Color-Indexed Images", IEEE Transactions on Image Processing, VOL. 13, NO. 11, Nov. 2004
- Ching-Yu Yang, "Based upon RBTC and LSB Substitution to Hide Data ", Proceedings of the First International Conference on Innovative Computing, Information and Control (ICICIC'06) 0-7695-2616-0/06 , 2006 IEEE

- www.Wikipedia.org
- http://www.devx.com/dotnet/Article/22667
- http://www.ddj.com/184409517
- MSDN (Microsoft Developer Network) Library

Study of C# and .net

- Wrox Professional C# 2005

Wrox Beginning Visual C# 2005