

## Implementation of Improved Steganographic Technique for 24-bit Bitmap Images in Communication

Mamta Juneja, Parvinder Sandhu

Department of Computer Science and Engineering, Rayat and Bahra Institute of Engineering and Biotechnology,

V.P.O Sahauran, Tehsil Kharar, Distt. Mohali, Punjab- 140104, INDIA

91-098786-77624

er\_mamta@yahoo.com

### Abstract

Steganography is the process of hiding one file inside another such that others can neither identify the meaning of the embedded object, nor even recognize its existence. Current trends favor using digital image files as the cover file to hide another digital file that contains the secret message or information. One of the most common methods of implementation is Least Significant Bit Insertion, in which the least significant bit of every byte is altered to form the bit-string representing the embedded file. Altering the LSB will only cause minor changes in color, and thus is usually not noticeable to the human eye. While this technique works well for 24-bit color image files, steganography has not been as successful when using an 8-bit color image file, due to limitations in color variations and the use of a colormap.

This paper presents the results of research investigating the combination of image compression and steganography. The technique developed starts with a 24-bit color bitmap file, and then compresses the file by organizing and optimizing an 8-bit colormap. After the process of compression, a text message is hidden in the final, compressed image. Results indicate that the final technique has potential of being useful in the steganographic world. [Journal of American Science 2009:5(2) 36-42] ( ISSN: 1545-1003)

**Keywords:** Bitmap, Colormap, Compression, LSB Based Insertion, Steganography

### Introduction

Steganography is an ancient technology that has applications even in today's modern society. A Greek word meaning "covered writing," steganography has taken many forms since its origin in ancient Greece. During the war between Sparta and Xerxes, Dermeratus wanted to warn Sparta of Xerxes' pending invasion. To do this, he scraped the wax off one of the wooden tablets they used to send messages and carved a message on the underlying wood. Covering it with wax again, the tablet appeared to be unused and thereby slipped past the sentries' inspection. However, this would not be the last time steganography would be used in times of war.

In World War II, the Germans utilized this technology. Unlike the Greeks, these messages were not physically hidden; rather they used a method termed "null ciphering." Null ciphering is a process of encoding a message in plain sight. For example, the second letter of each word in an innocent message could be extracted to reveal a

hidden message.

Although its roots lay in ancient Greece, steganography has continually been used with great success throughout history. Today steganography is being incorporated into digital technology. The techniques have been used to create the watermarks that are in our nation's currency, as well as encode music information in the ever-popular mp3 music file. Copyrights can be included in files, and fingerprints can be used to identify the people who break copyright agreements [5] [6] and [8]. However, this technology is not always used for good intentions; terrorists and criminals can also use it to convey information. According to various officials and experts, terrorist groups are "hiding maps and photographs of terrorist targets and posting instructions for terrorist activities on sports chat rooms, pornographic bulletin boards, and other Web sites"[1].

This aspect of steganography is what sparked the research into this vast field [3] and [4]. Education and understanding are the first steps toward

security. Thus, it is important to study steganography in order to allow innocent messages to be placed in digital media as well as intercept abuse of this technology [7].

### Current Usage

In modern society, steganographic techniques are used in a wide range of applications. Watermarking, a subclass of steganography, includes everything from watermarks on paper and security marks on paychecks to the hidden image of Abraham Lincoln on the five-dollar bill. Its purpose is to verify authenticity, prevent counterfeiting, or to give information about the creator. These are not the limits of steganography; it is also used in much the same manner in the digital world. Programs such as Adobe Photoshop version 4.0 or higher have the capability to embed and detect watermarks, while other programs like Stego can hide encrypted text messages in GIF images.

As hidden messages are being found and algorithms cracked, new methods of steganography are being developed. There has been a significant surge of interest since the first academic conference on the subject, the Workshop on Information Hiding, was organized in 1996 and held at the Isaac Newton Institute in Cambridge. One reason for this surge of interest is that the publishing and broadcasting industries have become interested in techniques for hiding encrypted copyright marks and serial numbers in digital films, audio recordings, books, and multimedia products. The new market opportunities due to digital distribution coupled with the fear that digital products could be too easy to copy leaves these industries in a rather large dilemma. Other motivations to study steganographic methods are due to governmental restrictions that have been placed on encryption services. People desire a way to send private messages. The controversial issue concerning this topic is that spies and criminals wanting to pass on secrets in inconspicuous data over public networks can abuse the same applications that are being developed to allow copyrights to be placed in digital media.

### Least Significant Bit Insertion

One of the most common techniques used in steganography today is called least significant bit (LSB) insertion. This method is exactly what it

sounds like; the least significant bits of the cover-image are altered so that they form the embedded information. The following example shows how the letter A can be hidden in the first eight bytes of three pixels in a 24-bit image.

```
Pixels:      (00100111 11101001 11001000)
              (00100111 11001000 11101001)
              (11001000 00100111 11101001)
```

```
A:   01000001
```

```
Result:      (00100110 11101001 11001000)
              (00100110 11001000 11101000)
              (11001000 00100111 11101001)
```

The three underlined bits are the only three bits that were actually altered. LSB insertion requires on average that only half the bits in an image be changed. Since the 8-bit letter A only requires eight bytes to hide it in, the ninth byte of the three pixels can be used to begin hiding the next character of the hidden message.

A slight variation of this technique allows for embedding the message in two or more of the least significant bits per byte. This increases the hidden information capacity of the cover-object, but the cover-object is degraded more, and therefore it is more detectable. Other variations on this technique include ensuring that statistical changes in the image do not occur. Some intelligent software also checks for areas that are made up of one solid color. Changes in these pixels are then avoided because slight changes would cause noticeable variations in the area [9] and [10].

While LSB insertion is easy to implement, it is also easily attacked. Slight modifications in the color palette and simple image manipulations will destroy the entire hidden message. Some examples of these simple image manipulations include image resizing and cropping [11] and [12].

### Bitmap File Structure

Before compression and steganographic techniques could be implemented, more information needed to be gathered about the bitmap file format. Microsoft defined the bitmap file format. In hopes of making this a popular file format, Microsoft made structures so that the information can be extracted easily from the bytes of a file. Bitmaps have two headers that contain the information needed to extract and display the image.

The first 14 bytes of the file make up the first header, which can be read into a structure called the `BitmapFileHeader`. The information contained in this structure can be seen in Table 1.

**Table 1: Bitmap File Header Structure Items**

| Item Name                    | Size    | Description                         |
|------------------------------|---------|-------------------------------------|
| <code>bfType</code>          | 2 bytes | ASCII for "B" and "M"               |
| <code>bfSize</code>          | 4 bytes | Size of the File in bytes           |
| 1<br><code>bfReserved</code> | 2 bytes | Reserved Equals 0                   |
| 2<br><code>bfReserved</code> | 2 bytes | Reserved Equals 0                   |
| <code>bfOffBits</code>       | 4 bytes | Number of Bytes to the Picture Data |

The next header in the bitmap can be read into a structure called the `BitmapInfoHeader`. The length of this header is determined by the first four bytes in the header, the `biSize` field. Generally, the length of the header is 40 bytes long. Table 2 shows the items that are contained in this structure.

**Table 2: Bitmap Information Header Structure Items**

| Item Name                    | Size    | Description                                   |
|------------------------------|---------|---|
| <code>biSize</code>          | 4 bytes | Size of the Bitmap Info Header in bytes       |
| <code>biWidth</code>         | 4 bytes | Width of the Bitmap in pixels                 |
| <code>biHeight</code>        | 4 bytes | Height of the Bitmap in pixels                |
| <code>biPlanes</code>        | 2 bytes | Number of Planes (Equals 1)                   |
| <code>biBitCount</code>      | 2 bytes | Number of Bits Per Pixel (1,4,8,16,24, or 32) |
| <code>biCompression</code>   | 4 bytes | Type of Compression (0,1,2)                   |
| <code>biSizelImage</code>    | 4 bytes | Size of the Picture Data in bytes             |
| <code>biXPelsPerMeter</code> | 4 bytes | Horizontal Resolution (Pixels/Meter)          |
| <code>biYPelsPerMeter</code> | 4 bytes | Vertical Resolution (Pixels/Meter)            |
| <code>biClrUsed</code>       | 4 bytes | Number of Actual Colors Used                  |
| <code>biClrImportant</code>  | 4 bytes | Number of Important Colors (0 = all)          |

An optional colormap follows the `BitmapInfoHeader` in the bitmap image file. Each color in the colormap is four bytes long and is contained in a `RGBQUAD` structure. Table 3 shows the items within the `RGBQUAD` structure. The program determines whether or not a colormap

exists by examining the `bfOffbits` field in the `BitmapFileHeader`. If this field is greater than the total size of the two headers, then a colormap is present.

**Table 3: RGBQUAD Structure Items**

| Item Name                | Size   | Description       |
|--------------------------|--------|-------------------|
| <code>rgbBlue</code>     | 1 byte | Blue Intensity    |
| <code>rgbGreen</code>    | 1 byte | Green Intensity   |
| <code>rgbRed</code>      | 1 byte | Red Intensity     |
| <code>rgbReserved</code> | 1 byte | Unused (Equals 0) |

A colormap is used in different ways depending on the `biBitCount` field in the `BitmapInfoHeader`. If the `biBitCount` equals 24, the colormap is used to list the most important colors of the image, but the colormap is not actually used by the picture data. If the `biBitCount` equals 8, each pixel in the picture data is an 8-bit pointer to a color in the colormap. A colormap in an 8-bit bitmap has a maximum 256 colors since the pointer into it can only be 1 byte ( $2^8 = 256$ ). The `biBitCount` field can be other values than 8 or 24, but since these are uncommon the current project does not deal with them.

Following the colormap (if a colormap exists) is the picture data. The picture data also varies depending on the value of `biBitCount`. If the `biBitCount` equals 24, then each pixel is represented by three bytes. There is exactly one byte for each intensity: red, green, and blue. If the `biBitCount` equals 8, each pixel is represented by 8-bits, which point to a `RGBQUAD` structure in the colormap.

One complication to the picture data is that each line of pixels in a picture, known as a scanline, must start on a LONG boundary. Since a LONG is four bytes in length, a LONG boundary is an address that is evenly divisible by four. The pixels in the image formats being discussed here are either 1 byte (8-bit images) or 3 bytes (24-bit images) long in the picture data. The number of bytes needed to accurately display a scanline may not be divisible by four. Since images can be any number of pixels wide, the end of the scanline may not end on a LONG boundary. In such cases, extra bytes of value zero are padded to the end of the scan line in order to make the scan line end on a LONG boundary. The next scanline can then start

on the LONG boundary as well.

### Proposed Technique

After reviewing the current products on the market for steganography, it was determined that there was not a practical implementation for 8-bit images. Although network speed is increasing, and bandwidth problems are decreasing, file size is still of utmost importance and smaller file sizes are optimal in network communication. Thus, the current steganographic use of 24-bit images leads to slower communication and development of an 8-bit image format would be beneficial.

The aim of this research is to create a practical steganographic implementation for 8-bit images. A 24-bit bitmap image would be converted to an 8-bit bitmap image while simultaneously encoding the desired hidden information. An algorithm would be created to select representative colors out of the 24-bit image to create the palette for the 8-bit image. This palette would then be optimized to an 8-bit colormap that could be applied with minimal changes to the quality of the original image.

This process of compressing the image from a 24-bit bitmap to an 8-bit bitmap resulted in minor variations in the image, which are barely noticeable to the human eye. However, these slight variations aid in hiding the data. Since there would not be an original 8-bit image to compare with the stego-image, it would be impossible to discern that the slight variations caused by hiding the data are different from the slight variations caused by compression.

A practical steganographic implementation for 8-bit images enabled smaller file sizes to be utilized in steganographic communications. While also limiting the size of the hidden file, this implementation addressed issues that have been passed by in other applications, and provided a more compact vehicle for those secret communications that do not require a large cover-file.

### Creating the Colormap

The colormap in an 8-bit color image has a maximum of 256, 24-bit colors. However, in order to minimize the noise added when the least significant bits are changed, a starting colormap of

only 240 colors is created. Sixteen additional colors will be added to the colormap by the time the final picture is written.

In order to select the 240 original colors, the image is divided into a grid of fifteen quadrants by sixteen quadrants, as seen in Figure 1. One color is chosen from each of these quadrants by randomly selecting a set of X and Y coordinates within each quadrant. Calculations are then made to determine the index of the pixel in the array of RGBQUADS that represent the image data. (An RGBQUAD is a structure containing four bytes, one each for the red, green, and blue intensity and a reserved byte.)

Each time a color is selected from a quadrant, it is compared to every other color in the colormap, and the minimum error between any two colors is calculated. If this error is lower than a certain error level (currently set at 20), then the new color is discarded and another color is selected from the image. After five attempts to find a color from a certain quadrant that differs enough from all the other colors in the colormap, the selected color is added to the colormap and the program moves to the next quadrant.

### Optimizing the Colormap

The initial colormap contains 240 colors that were picked out of the original image. These colors were chosen from the entire image but that does not guarantee that these colors are the most representative of the colors that exist in the image. Therefore, the colormap must then be optimized to provide the best 240 colors for the colors in this particular image.

The optimization algorithm uses a Linde-Buzo-Gray methodology<sup>3</sup>. A pixel is chosen from the original 24-bit image and its RGB values are compared to the RGB values of every color in the colormap. For each comparison an error level is calculated using the mean absolute error of the red, green, and blue component of the color. The colormap color that produces the smallest amount of error is the colormap color that is closest to the

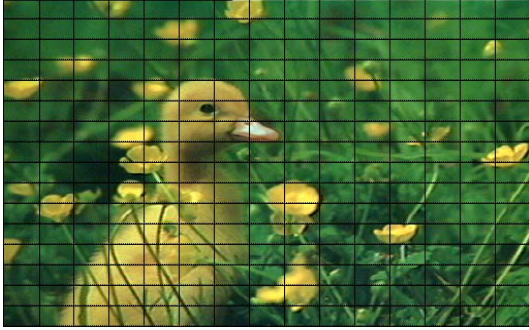


Figure 1: Image overlaid with 15x16 grids

pixel's RGB values. The RGB values of the pixel are then added to the RGB values of the colormap color. A counter is implemented to keep track of how many pixels are assigned to each colormap color and is incremented each time a match is found. Approximately 25% of the pixels in the original image are run through this process. Once the algorithm has gone through the whole image, the RGB values of each colormap container are averaged by dividing the red, green, and blue values by the counter for that particular container. This process produces a colormap with slightly new, "better" colors in it. The process is repeated until the new colormap is considered to be optimized. To determine when the colormap is optimized, the error levels are recorded during each run and when a certain error level is attained the algorithm is finished.

### Sorting the Colormap

Each pixel in an 8-bit color image is an 8-bit pointer to a 24-bit color in the colormap. Looking ahead to the LSB insertion, a pixel pointing to a red color could suddenly point to a yellow color by a simple flip of the least significant bit. In order to reduce dramatic noise such as that, the colormap was sorted so that similar colors are next to each other before the pixels are assigned to colormap colors. The sorting algorithm works as follows.

Beginning with the first color in the colormap array, the pixel that is the closest in color to the starting pixel is found using the mean absolute error measure. If the best match to a color results in an error level greater than 100 (meaning that there really was not a very good match to the color), a new color is created in the first open slot (using the sixteen extra spaces in the colormap) and this new color is used as the pair. The best-matched

color is then switched with the color immediately following the starting color. The same procedure is repeated with the next color that has not been matched. Once the original 240 colors have been matched, additional colors are created to fill any of the extra sixteen positions left in the original colormap.

### Assigning Pixels

After sorting the colormap, the 8-bit image is almost ready to be created. An 8-bit bitmap contains a colormap of 256 colors and contains an assignment of each pixel to a color in the colormap. To assign the pixels to a colormap color, the original 24-bit image pixels are used. A pixel is chosen from the original 24-bit image and its RGB values are compared to the RGB values of every color in the colormap. For each comparison an error level is calculated using the mean absolute error of the red, green, and blue color components. The colormap color that produces the smallest amount of error is the colormap color that gets assigned to this pixel.

### Encoding the Data

The image is now ready to have data embedded into it. The encode function takes three parameters and two steps in order to complete. The data string of text, picture data, and binary data string are the three parameters for the first step in encoding the text into the image. The first step in the encoding function is to convert the ASCII text into its binary equivalent. In order to do this, each character of the text message is converted to its ordinal number (example: 'a' = 97). The ordinal number is then converted to binary using the following method called the division-remainder routine. An ordinal number is divided by two using the *mod ()* function. This function returns either a one or a zero, which is then placed in a remainder array. This is continued until the dividend is zero. The ones and zeros in the remainder array is the binary equivalent of the ASCII ordinal number. Then, once all characters have been converted in this fashion, the binary data is embedded in the image by sequentially altering the least significant bit of the image data as necessary.

### Additional features

Since LSB insertion is easily attacked, repetition of the hidden message throughout the entire image will improve the integrity of the message. For example, if the image were cropped with the message embedded throughout the entire image, chances are good that the message could still be extracted from the cropped image. Repetition of the hidden message also causes the image quality to be uniform throughout instead of having noticeable areas of variations in certain sections of the image.

An encryption algorithm can also be added to this to further protect the end user's message if so desired. While embedding a message in an image will prevent most people from knowing a message exists, encrypting the message before embedding it will aid in safeguarding it from being read by someone who happens to find the message.

The encryption technique employed is a 24-bit algorithm. The user enters a number between 0 and 16,777,216 to use as the encryption key, which is converted to a 24-bit representation. The first three bytes (twenty-four bits) of the message are XORed with the 24-bit key. The next three bytes of the message are then XORed with the 24-bit key. This process is continued in a similar fashion for the remainder of the message. The following example shows how the algorithm would encrypt a 6-character message.

```

Message = 'abcdefg'
Key = 9,657,852
'abc' = 01100001 01100010 01100011
'def' = 01100100 01100101 01100110
9,657,852 = 10010011 01011101 11111100

resultabc = 'abc' XOR 9,657,852
           = 011000010110001001100011 XOR
           100100110101110111111100
           = 111111100011111110011111

resultdef = 'def' XOR 9,657,852
           = 011001000110010101100110 XOR
           100100110101110111111100
           = 111101110011100010011010

Encrypted Message =
111111100011111110011111110111001110001
0011010

```

### Conclusion

The purpose of this research is to investigate steganographic techniques, and apply them to 8-bit images. While this research does not cover every

aspect of steganographic technology, it is a good starting point for anyone interested in implementing and learning how the technology can be applied to digital images. Further developments to this could include using different picture formats, such as JPEG and GIF, to serve as the cover object. Additional features could allow other files such as other pictures or MS Word documents to be used as hidden messages. Other future endeavors could employ some of the variations of least significant bit insertion such as using two or more of the least significant bit to embed the data. This will yield a higher embedding space but also increase image degradation levels. To counteract this, the sorting of the colormap would need to be adjusted based on the number of bits used in the LSB insertion algorithm

### Correspondent Author:

Mamta Juneja  
Assistant Professor, CSE Department,  
RBIET, Saharan-mohali, Punjab, INDIA  
Email:er\_mamta@yahoo.com  
Phone: +91-98786-77624

### References

- B.Schneier, "Terrorists and Steganography", 24 Sep.2001,available: <http://www.zdnet.com/zdnn/tories/comment/0,5859,2814256,00.html>.
- Y. Linde, A. Buzo, and R. M. Gray, "An Algorithm for Vector Quantizer Design," *IEEE Transactions on Communications*, pp. 84-95, January 1989.
- Andersen, R.J., Petitcolas, F.A.P., On the limits of steganography. *IEEE Journal of Selected Areas in Communications, Special Issue on Copyright and Privacy Protection* **16** No.4 (1998) 474-481.
- Johnson, Neil F. and Jajodia, Sushil. "Steganography: Seeing the Unseen." *IEEE Computer*, February 1998, pp.26-34.
- William Stallings; *Cryptography and Network Security: Principles and Practice*, Prentice Hall international, Inc.; 2002.[2]
- Eric Cole,"Hiding in Plain Sight: Steganography and the Art of Covert Communication"

Gregory Kipper, "Investigator's Guide to Steganography"  
Stefan Katzenbeisser and Fabien, A.P. Petitcolas,"  
Information Hiding Techniques for Steganography  
and Digital Watermarking"

Hiding secrets in computer files: steganography is  
the new invisible ink, as codes stow away on  
images-An article from: The Futurist by Patrick  
Tucker

Ismail Avcibas,, Member, IEEE, Nasir  
Memon,Member, IEEE, and Bülent Sankur,  
Member, "Steganalysis Using Image Quality

Metrics," IEEE Transactions on Image Processing,  
Vol 12, No. 2,February 2003..

Niels Provos and Peter Honeyman, University of  
Michigan, "Hide and Seek: An Introduction to  
Steganography" IEEE Computer Society IEEE  
Security &Privacy.

R. Chandramouli and Nasir Memon, "Analysis of  
LSB Based Image Steganography Techniques",  
IEEE 2001