ABSTRACT

This article presents a new method for hiding data within existing image data. The technique falls into the category of steganography, which is an approach of hiding data within other data sets. This new algorithm is named StegBlender, and is a new methodology for hiding data. It is different than traditional steganography methods, and should become part of the standard techniques used.

In this article, a bit shaving technique which has been used extensively is explained in order to give a context. Then, the StegBlender algorithm is explained as a contrast, including advantages over traditional techniques.

KEYWORDS

Steganography, blend, data hiding, obfuscation, embedded data.

1. Introduction

One of the tasks of security professionals is to protect the confidentiality of data. The most common and effective technique of insuring data confidentiality is with data encryption. Encryption converts normal data to what is known as cyphertext. It is a reversible process, and the plain, unencrypted data can be recovered with a process that reverses the encryption. While encryption provides good confidentiality, there is another way to provide confidentiality known as steganography. It is a method of hiding data so that nobody even knows it exists [1]. In this way, the data is not subject to encryption attacks because nobody knows that the data exists.

In order to hide data, you must have a carrier file into which the data will be hidden. The new technique that this article presents uses image files as carrier files. Specifically, BMP files are used. The algorithm presented could easily be extrapolated to use on PNG files. That is not to say that other image formats such as JPG, GIF, and TIF cannot be used, they would each need to take a slightly different approach since their data is laid out differently than BMP and PNG images. But to make the concepts clearer, only application of steganographic techniques for BMP will be undertaken in this article. Applying steganography to JPG would be a valuable addition to the literature, and that is in the planning stage.

2. Bit Shaving – A Common Steganographic Technique

Before delving into a new method, it would be best to discuss one of the more common steganographic techniques in use today which use image files as carriers. It is best described as bit shaving because it sets at least a single bit from the carrier data bytes to a zero value. Most of the time, the least significant bit is set to zero for each byte in the carrier data. For instance, if a carrier byte has the binary value of 10110111 then the least significant bit would be set to zero, and the new value would be 10110110. The only difference between the first byte and the altered byte is that the least significant bit becomes a zero after the alteration. Table 1 illustrates the process of shaving bits from values with several examples. Please note that the resulting decimal value does not always change when the least significant bit is shaved.
### Table 1: Decimal and Binary Values Before and After Bit Shaving

<table>
<thead>
<tr>
<th>Decimal Values</th>
<th>Binary Values</th>
<th>Shaved Binary Values</th>
<th>Resulting Decimal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>187</td>
<td>1 0 1 1 1 0 1 1</td>
<td>1 0 1 1 1 0</td>
<td>186</td>
</tr>
<tr>
<td>87</td>
<td>0 1 0 1 0 1 1 1</td>
<td>0 1 0 1 0 1</td>
<td>86</td>
</tr>
<tr>
<td>214</td>
<td>1 1 0 1 0 1 1 0</td>
<td>1 1 0 1 0 1</td>
<td>214</td>
</tr>
<tr>
<td>119</td>
<td>0 1 1 1 0 1 1 1</td>
<td>0 1 1 1 0 1</td>
<td>118</td>
</tr>
</tbody>
</table>

### 3. Image Data

First, though, a few words about image data are in order. Most image data, especially for the purposes of this article, are sets of three bytes. These sets of three bytes represent single screen pixels. The three bytes each represent a different color channel. One is for red, one is for green, and one is for blue. Mixing these three color channels provides the range of colors that we see on modern computer displays. Each byte has a value ranging from 0 to 255. The greater the channel value, the greater the effect of that color channel on the resulting color.

We normally refer to the three bytes as RGB triples, standing for Red, Green, and Blue triples. If an RGB triple has the value of 255, 0, 0 then the red value is at maximum, and the green and blues values are at minimum. This results in a pure red pixel on the computer display. In the same manner, an RGB triple of 0, 255, 0 yields a pure green color while 0, 0, 255 yields a pure blue color. Many images contain a fourth byte which indicates an alpha value. This allows for pixels in an image to have varying degrees of transparency. To keep this article clear, only images without alpha values will be considered. Fig 1 shows some common RGB triples and their resulting colors.

### 4. Hiding a Single Byte of Data

For this discussion, only one bit will be shaved from each carrier byte, although it could be more than a single bit. In order to hide one incoming bit then, we need to shave the least significant bit from one carrier byte. Once the least significant carrier byte has been shaved and is zero, the incoming bit is placed into the position of the least significant bit. Figs 2 through 4 illustrate the process. Each figure uses eight carrier bytes of image data to hide eight bits from the data that is to be hidden.

![Figure 1: Common RGB Colors](image1.png)

**Figure 1: Common RGB Colors**

**Figure 2: Here you see eight carrier bytes: 76, 123, 205, 82, 129, 211, 245, and 255. You also see their binary equivalents. Notice that the least significant bits of each carrier byte are a different color in the figure, and indicate that they must all be shaved, or set to zero.**

![Figure 3: The least significant bits of each carrier byte have been set to zero.](image3.png)
Figure 4: Each bit of the character ‘A’ (decimal 65, binary 01000001) is placed into the least significant bit of the carrier.

Note that each carrier byte holds one message bit. For this reason, we can calculate the amount of carrier bytes needed to hide any given message by multiplying the size in bytes of the message by eight. For instance, a message of 20 bytes is 160 bits. It therefore requires 160 carrier bytes to be hidden, or eight times the message size of 20.

The entire process of bit shaving and replacement can be seen in Fig 5.

4. Acceptable Data Degradation

It should be obvious that the carrier data which now contains the message is not exactly the same as the original carrier data. From this, the question arises “can a viewer of the new image detect the difference between the original image and the altered image?” The answer is no for the following reasons.

First, each color channel has a value ranging from 0 to 255. This means that if the least significant bit is set to zero, it only affects the color channel value by .39 percent. And by the law of averages, the least significant bit is only one half of the time, while the other half of the time it is already a zero. So the net result is a change in a color channel of .39 percent half of the time and a result of 0 percent the other half of the time. Averaged over an entire image, this produces a statistical change of .195 percent change, even less than the .39 percent change calculated.

There is a branch of psychology that mentions the amount of change needed in order to be noticed. The concept is known as Just-Noticeable Difference[2]. It says that there is a threshold that must be exceeded in order for a change to be perceived. The general number of 1.25 percent is widely accepted, and is sometimes referred to as Weber’s fraction, or Weber’s constant. This leads to the conclusion that the .39 percent resulting when shaving the least significant bit is less than the threshold needed for a change in perception.

In order to demonstrate this, a program can be used to compare an original color with a color that has at least one bit shaved. Fig 6 shows an original color and a color with the least significant bit shaved. The change in the viewable color is not perceptible. This program was written especially for this article, and can be found at https://github.com/RickLeinecker/StripColorBits.

Figure 6: This color can be seen in its original hue in the rectangle to the left. With a single bit shaved, the new color can be seen in the second rectangle. There is no perceptible difference between the original color and the color that has had the least significant bit shaved.

5. Bit Shaving and Replacement Detection

There are tools available to detect steganographically hidden messages in images. They are not very effective because the detection tool must know the exact methodology that was used, and then reverse it to retrieve the message.

For the bit shaving and replacement method, there is one giveaway that could tip off a
detection tool. Suppose that before a message was hidden, the entire carrier had each byte’s least significant bit shaved. Now the entire image data has the least significant bit set to zero, a statistical improbability. Then suppose that the hidden message only occupies the least significant bits of the first half of the carrier bytes. This means that the entire second half of the carrier still has all least significant bits as zeros, another improbability. A competent detection program would conclude that there was a steganographic message hidden in the carrier bytes.

There are two ways to mitigate this detection hazard. The first is to only shave the least significant bits when there is message data to hide. The other is to go through the unused least significant bits and randomly set some of them to one in order to avoid detection.

6. A New Method of Hiding Data in Image Data

The most important factor in keeping steganographically hidden data hidden is variety[3]. If detection software knows exactly how a message was hidden, it can easily retrieve it. For that reason, a different method was developed. Not only does it have a different methodology to hide data from what this article has discussed, but it is parameterized and can change from use to use. This greatly reduces the ability to detect and retrieve a hidden message. The name of this method is StegBlender.

While the bit shaving technique alters the least significant bit of each carrier byte, StegBlender alters unknown bits in a sequence of carrier bytes. Not knowing which bits have been altered makes it much more difficult to detect hidden messages.

The first concept to define is that of the carrier byte group. This is a grouping of a number of carrier bytes. In bit shaving, there was always a single carrier byte that was acted on for each operation. But with StegBlender it is a group of carrier bytes, and the number of carrier bytes in a group can change from carrier to carrier, thus providing some of the variety.

Once an image has been loaded into a memory buffer, the algorithm moves a pointer from the start of the buffer to the end of the buffer, or as long as necessary to hide the message data. After the StegBlender algorithm is finished with the current group of carrier bytes, the pointer moves up to the next group of carrier bytes. Fig 7 shows the process of moving from group to group.

Figure 7: The image data is divided into groups of carrier bytes. After the algorithm finishes with one group, it moves to the next group.

For each group of carrier bytes a total must be calculated. The algorithm relies on having a total of the current set of carrier bytes. This is done by looping through each one of the carrier bytes and adding their values to an accumulator variable. Pseudo code for a function named `countTotal` that does this is shown in Listing 1, while Fig 8 depicts it as a flowchart.

```plaintext
Function countTotal
Parameters: array data, number groupSize
Start
i = 0, total = 0
i < groupSize
Stop
FALSE
total = total + data[i]
i = i + 1
return total
```

Listing 1: The countTotal function which calculates the total of all values within a group of carrier bytes.

7. Data Group Mod Value
The magic of the StegBlender algorithm isn’t the total of the carrier byte groups, but of a derived value from that total and a modulus value. In order to find the hidden value of a group, the modulus of the total of that group and a predetermined modulus value is calculated. For instance, if the predetermined modulus value is 256 and the total of a group is 1234, then the resulting value after performing the modulus operation is 210. The following formula shows this.

\[ \text{hiddenValue} = \frac{\text{groupTotal}}{\text{predeterminedValue}} \]

\[ \text{hiddenValue} = 1234 \mod 256 = 210 \]

In other words, the algorithm depends on calculating the modulus value of the total of each group of carrier bytes. The modulus value, when the carrier bytes in the group are adjusted, will equal the next message byte that is to be hidden.

For instance, let us suppose that the message that must be hidden is the set of four characters TEST. Those four characters have ASCII values of 84, 69, 83, and 84. Since there are four bytes to hide, the algorithm requires four carrier byte groups. Ideally the first group will have the mod value of 84, the second 69, the third 83, and the fourth 84. The pseudo code in Listing 2 shows how to determine the hidden values in carrier bytes, the function being `extractMessage`, and Fig 9 shows the flowchart.

```python
predeterminedModValue = 256
extractMessage array data, number groupSize
    create outBuffer
    while not done
        total = countTotal data, groupSize
        messageValue = total mod predeterminedModValue
        append outBuffer, messageValue
        data = data + groupSize
        if exit condition exists
            then done = true
    return outBuffer
Listing 2: The extractMessage function determines the hidden message.
```

8. Hiding a Message

It is easier to extract a message than to hide a message. That is because when a message is extracted, no adjustment of the group of bytes is necessary. But for each message byte that must be hidden, the bytes in the current group must be adjusted until the modulus value of the group’s total matches the message byte.

Let us suppose that the message byte that must be hidden in this group is 84. Now let us suppose that the modulus operation value does not match that. In order to adjust the modulus operation value, we go through the carrier bytes and either add or subtract one until the modulus operation value matches the message byte.

As a special note, if the group total is above the halfway point, meaning that the average of the group bytes is more than half of the predetermined modulus value, then the algorithm subtracts. On the other hand, if the group total is below the halfway point then the algorithm adds. This ensures that the carrier bytes stay within range without an overflow. The pseudo code in Listing 3 shows the process of adjusting the contents of a group so that its modulus operation value matches the byte that is to be hidden. Fig 10 shows the flowchart for the adjustment code.

```python
getModOperationValue array data, number groupSize
    total = countTotal data, groupSize
    return total mod predeterminedModValue
calcDelta array data, number groupSize
```

Figure 9: The extractMessage function flowchart.
total = countTotal data, groupSize
halfway = groupSize * predeterminedModValue / 2
if total > halfway then return -1
else return 1

adjustGroup array data, byte messageValue
circular = 0
delta = calcDelta data, groupSize
while getModOperationValue data, groupSize <> messageValue
data[circular] = data[circular] + delta
circular = circular + 1
if circular >= groupSize then circular = 0

Listing 3: The adjustGroup function adjusts the carrier bytes in the group until the modules operation value matches the message byte.

minimumCarrierSize = messageSize x groupSize

The group size plays a role in the quality degradation of the carrier image. The larger the group size, the less each byte in the group must be altered to arrive at the correct modulus value. On the other hand, larger group sizes may not leave enough room into which the message can be hidden.

The predetermined modulus value can be adjusted in many cases, especially for ASCII values. Since ASCII values range from 0 to 128, the modulus value could be as low as 129.

10. Source Code Project

The source code for the entire project is posted on GitHub at the address https://github.com/RickLeinecker/StegBlender/. The source code should easily compile in any environment.

11. Conclusion

StegBlender represents a new approach to steganography. It is an algorithm that provides a way to hide information while evading detection. Two of its integral values, the group size and the predetermined modulus value, can be changed. With changeable values and with the fact that the addition and subtraction alters unpredictable bits, detection is very difficult.

Future research should include a version that embeds messages in Jpeg images. The reason this is important is because Jpeg is such a pervasive image format. This research is currently underway, and will be completed after the publication of this article.

It is hoped that contributors to the project which has been posted on GitHub will move this research forward, and to provide valuable feedback on the extant research.

REFERENCES
