LeonSteg

Steganography with sole 1 bpp modification

Left image (6906x4609px, JPEG format) is encrypted and stored bitwise in the middle steganography image (3888x2592px, PNG format). Right image shows the difference map between the initial and the steganography image. For perceptible visualization the difference values have to be mapped from range 0-1 to range 0-255.

Frank Nagl[1], Paul Grimm[2]

[1] Troonie Project eMail: info@troonie.com, http://troonie.com
[2] Fulda University of Applied Sciences eMail: paul.grimm@cs.hs-fulda.de, www.hs-fulda.de

Abstract This paper presents LeonSteg, a novel steganography approach, which directly encrypts and stores payload within image content. In this context exclusively one channel byte value of a pixel will be modified only in its last bit. Hence the storing is done by the smallest possible modification of a pixel. Encryption and choice of pixel are determinated highly dynamic constrained by one-way functions and hash functions. A final obfuscation process obscures the possibility to identify the modified pixel. LeonSteg is independently of color depth and number of channels of a pixel.

1 Introduction

Steganography conceals an information within a file, e.g. an image. In contrast to conventional cryptography, steganography deals with concealing the fact that a secret information is existent as well as concealing the content of the secret information [10]. Hence a steganography process is only as good as it is satisfying these criteria.
LeonSteg is a steganography approach, which firstly encrypts and secondly stores the payload within the image content by **the smallest possible modification** of a pixel. Encryption and choice of pixel are determinated highly dynamic constrained by one-way functions and hash functions.

2 State-Of-The-Art

In the past already several publications and tools were developed in the research area of image-based Steganography. In 1995 and 1998 Neil F. Johnson gave an overview about the history of (image-based) Steganography [2][3]. In addtion, digital data hiding techniques for images were analyzed, explored, countered and attacked [6]. In 2010 several steganography tools were introduced and evaluated on the weblog Heise Security [8]. In [1] a steganography approach works by selective hard disc fragmentation. Latest publications presented mobile cameras as goal for steganographic algorithms [7] and described the detection of URLs in image-based Steganography mechanism [4]. In contrast to other publications and tools, the LeonSteg steganography is constructed highly dynamic constrained by one-way functions and hash functions.
The author of this paper presented in 2016 **Steghash** [5], also a steganography approach. Even the basic idea of both approaches are totally different, LeonSteg is a kind of improvement and further development of Steghash. A comparison is shown in subsection 4.3.

3 Approach

LeonSteg encrypts and stores an initial byte array within image content directly without visible or qualitative decreases and independently of color depth and number of channels of a pixel.

3.1 Basic Idea

The basic idea of LeonSteg is to use the eight bits of a byte and store every bit in a several pixel. In this context exclusively one of all channel values (e.g. three in common RGB color images) needs to be modified. If a bit is occupied by 1, a bitwise OR operation with expression 00000001_B on the channel is done. Eq. 1 shows an example while the channel value of the pixel is 128_D.

$$\begin{aligned} 10000000_B &\mathrel{\widehat{=}} 128_D \\ \vee\, 00000001_B &\mathrel{\widehat{=}} \quad 1_D \\ \hline 10000001_B &\mathrel{\widehat{=}} 129_D \end{aligned} \tag{1}$$

If the bit is occupied by 0, a bitwise AND operation with expression 11111110_B is done. Eq. 2 shows this operation.

$$\begin{aligned} 10000000_B &\mathrel{\widehat{=}} 128_D \\ \wedge\, 11111110_B &\mathrel{\widehat{=}} 254_D \\ \hline 10000000_B &\mathrel{\widehat{=}} 128_D \end{aligned} \tag{2}$$

Either the channel byte value is **modified only in its last bit or no change** is done. Hence the storing is done by **the smallest possible modification** of a pixel.

3.2 Declaration Of Algorithm Parameters

This subsection declares some parameters for explaining in consecutive text the LeonSteg algoritm more clear. Moreover the paper contains snippets of source code borrowed from the implementation (see subsection 4.1). As initial values two parameters are given, the unencrypted **bytes[]** array and the **key** as passphrase string. Three further arrays need to be declared: **bits[]** containing all encrypted bytes as 8 (bit) boolean values, **usedPixel[]** storing information, which pixel are already used (including information which channel is used and whether the channel value is changed or not) and **hash[256]**, containing a hash byte array with exact 256 elements (see subsection 3.3). Moreover, three indices need to be declared: **indexPixel** indexing the current pixel in LeonSteg process, **indexChannel** indexing the RGB channel of the indexed pixel and **indexHash** indexing the current hash element using for en-/decryption as well as determining the next pixel to use.

3.3 Usage Of The Hash Array

A cryptographic hash function is used to map the **key** passphrase string to a hash byte array with exact 256 elements. This **hash[]** array is used in several context. The usage of an hash element is always combined with the modification of the element itself as well as **indexHash** and **indexChannel**, shown in listing 1. **indexChannel** is computed by applying the modulo operation of the hash element (dividend) and the number of channels per pixel (divisor). **indexHash** is simply added with the hash element and 1. In contrast to the other indices (common integer) the datatype of **indexHash** is byte, obviously with the range 0–255. An intended overflow ensures that the index accurately fitting with the **hash[]** array. The hash element itself (also byte as datatype) is added with its own sum of digits and also a potential overflow is intended. These modifications ensures a **dynamic behavior** in the encryption as well as in the complete steganography process. By adding the sum of digits to the correspondent hash element the **hash is changed continuously by a one-way function**.

```
byte GetAndTransformHashElement()
{
        // getting current hash element
        byte b = hash[indexHash];
        // channels per pixel
        int cpp = 3; // when grayscale cpp = 1
        indexChannel = b % cpp;
        // always change value after usage
        hash[indexHash] += DigitSumOfByte (b);
        // always increment additionally
        indexHash += (byte)(b + 1);
        return b;
}
```

Listing 1: Usage and modification of hash elements

3.4 Encryption And Decryption

Before the steganography process itself is started the initial **bytes[]** need to be encrypted. For encryption, the elements of the **bytes[]** array are added with hash elements, shown in listing 2.

```
// start values
indexHash = key.Length;
hash = GetCryptedHash (key);

// encryption by adding hash element
for (int i = 0; i < bytes.Length; i++) {
        bytes[i] += hash [GetAndTransformHashElement()];
}

// convert bytes array to bits array
bits = ConvertToBits(bytes);

// writing bits into image content
...
```

Listing 2: Encryption of initial bytes array

After encryption the **bytes[]** results in the **bits[]** array, which depicts the payload (the secret information) for the steganography process (see subsection 3.5). Clearly, for decryption the encrypted elements of **bits[]** array are read out first and in a second step the decryption process starts by subtracting the correspondent hash element as shown in listing 3.

```
// reading out bits from image content
...

// reset values
indexHash = key.Length;
hash = GetCryptedHash (key);

// convert bits array to bytes array
bytes = ConvertToBytes(bits);

// decryption by subtracting hash element
for (int i = 0; i < bytes.Length; i++) {
        bytes[i] -= hash [GetAndTransformHashElement()];
}
```

Listing 3: Decryption of encrypted bits

3.5 Steganography Process

After encryption the **bits[]** array is filled. In respect of the basic idea (described in subsection 3.1) all elements are stored in (one channel of) separate pixel. Hence the channel byte value will be modified only in its last bit or no change will be done.

Analog to the encryption the steganography process is done in a dynamic way through modifications of several parameter by using the **hash[]** array (see subsection 3.3). Listing 4 depicts the steganography process.

```
// reset values
indexHash = key.Length;
hash = GetCryptedHash (key);

// initialization local values
int indexPixel = 0;
int w = image.Width;
int h = image.Height;
int max = w * h;

for (int i = 0; i < bits.Length; i++) {
        indexPixel += GetAndTransformHashElement();

        // check, if indexPixel is out of range
        if (indexPixel >= max) {
                indexPixel -= max;
        }

        // get next not yet used pixel
        while (usedPixel [indexPixel].Used) {
                indexPixel++;
                // check again
                if (indexPixel >= max) {
                        indexPixel -= max;
                }
        }

        // get pixel position
        posY = indexPixel / w;
        posX = indexPixel - posY * w;

        // store bit by using basic idea
        bool changed = SetBitInPixel(bits[i], posX, posY,
           indexChannel);

        //mark pixel as used
        usedPixel[indexPixel] = new PixelInfo {
                Used = true,
                Channel = indexChannel,
                ValueChanged = changed
        };
}
```

Listing 4: Dynamic storing of bits into the image

The readout process is clearly almost identical as shown in listing 5.

...

```
for (int i = 0; i < w * h; i++) {
        indexPixel += GetAndTransformHashElement();
        ...
        // read out
        bool bit = GetBitInPixel(posX, posY, indexChannel);
        bits.Add(bit);

        //mark pixel as used
        usedPixel[indexPixel] = new PixelInfo {
                Used = true
        };
}
```

Listing 5: Readout bits from the image

3.6 Obfuscation

Even the dynamic behaviour in LeonSteg (caused by continuous changing by a one-way function of the **hash[]** array) a difference map of inital and corresponding steganography image would show the compromised pixel (respectively its compromised channel). For avoiding this recognition an obfuscation process finalizes the LeonSteg algorithm. Every pixel that was not used as well as every pixel that was not changed is modified in a none-used channel by changing its last bit as shown in listing 6. Thus a difference map between the initial and the steganography image shows a **modification in every pixel** (in one channel), even the pixel does not contain part of the payload or the payload is stored in another channel with a none-modified channel value (as shown in equation 2). Figure 1 and figure 2 show an initial and the corresponding steganography image as well as the corresponding difference map.

```
...

for (int i = 0; i < w * h; i++) {
   GetAndTransformHashElement();
   bool used = usedPixel [i].Used;
   bool changed = usedPixel [i].ValueChanged;
   int channel = usedPixel [i].Channel;

   if (!used || (used && !changed)) {
      if (used) {
         while (indexChannel == channel) {
            GetAndTransformHashElement ();
         }
      }

      byte b1 = GetChannelByte (posX, posY, indexChannel);
      byte b2 = (byte)(b1 | 1);
      bool bit = true;
      if (b1 == b2) {
```

```
            bit = false;
        }

        // store bit by using basic idea
        SetBitInPixel(bit, posX, posY, indexChannel);
    }
}
```

Listing 6: Obfuscation

4 Realization and Evaluation

4.1 Implementation Details

In the implementation four (static) bytes are added in the end of the initial **bytes[]** array. This is done to mark the end point for the steganography process.
As cryptographic hash function to compute the **hash[]** array, a loop of SHA-512 is used. Every turn produces an hash array with 64 elements. In the first turn the **key** passphrase string is used as initial data. In the second to fourth turn the resulting hash array of previous turn is used as initial data. By merging all four hash arrays the **hash[]** byte array with exact 256 elements is generated.
LeonSteg is implemented as part of the open source image processing and converting tool Troonie [9]. LeonSteg as well as Troonie are developed with .NET (Mono framework) and GTK#.

4.2 Evaluation

The maximum size of potential payload (in bytes) matches with the image resolution. In the implementation four bytes are needed to mark the end of the payload (mentioned in subsection 4.1). An image with the common resolution of 12 MP can store 1,44 MB. Equation 3 shows the computation.

$$\begin{aligned} max_{(width*height)} &= width * height/8 - 4 \\ max_{(12MP)} &= 4256 * 2848px/8 - 4 \\ &\widehat{=} 1515132\ B \widehat{=} 1,44\ MB \end{aligned} \tag{3}$$

Thus the resolution of a steganography image could be smaller than an image which is used as payload. In the teaser of this paper the left image (6906x4609px, JPEG format) is encrypted and stored in the steganography image (3888x2592px, PNG format) as shown in the middle image of the teaser.
In subsection 3.1 is explained that exklusively one channel byte value of a pixel is **modified only in its last bit or no change** is done. Hence the storing of payload is done by **the smallest possible modification** of a pixel. Furthermore the obfuscation process (see subsection 3.6) ensures a **1-bit-modification in every pixel** (in exklusively one channel). Figure 1 shows an initial image and the corresponding steganography image which look obviously identical. Only the difference map (as

Figure 1: Initial image (left) and corresponding steganography image (right). Both images look obviously identical.

shown in figure 2) discloses the (smallest possible) deviation of 1 bit between pixel (relating to all channels). Equation 4 defines the deviations of initial RGB (24 bpp) and grayscale (8 bpp) pixel relating to their correspondent steganography pixel.

$$\begin{aligned} \delta_{RGB} &= 1/(3 * 256) \;\widehat{=}\; 0,1302\% \\ \delta_{grayscale} &= 1/(1 * 256) \;\widehat{=}\; 0,3906\% \end{aligned} \tag{4}$$

4.3 Comparison with StegHash

In StegHash [5] the distance between two pixel containing payload is static. It is stored as **unencrypted plain value** as beginning part of the payload within the steganography process. In contrast to StegHash, LeonSteg does not store any distance values. These values are computed dynamically and in this way every pixel pair has a different distance.
Furthermore the deviation of every pixel from its original (independently pixel containing payload, just modified by obfuscation or both) is always $\delta_{RGB} = 0,1302\%$. In StegHash the value amounts minimum $\delta_{RGB} = 0,1302\%$, maximum $\delta_{RGB} = 1.172\%$ and on average $\delta_{RGB} = 0.586\%$. Consequently the deviation of LeonSteg is on average $4.5x$ **less**.
In addition the maximum size of potential payload of LeonSteg is directly scalable, whereas in StegHash it depends on several not scalable parameters. In average StegHash can store less then three times more payload than LeonSteg but with a higher deviation and a static and unencrypted distance value.

4.4 Payload Enlargement

To enlarge the maximum size of potential payload a modified version, called **LeonStegRGB**, is implemented. In contrast to LeonSteg not exklusively one channel but all channels of a pixel can be used. In common RGB colored photos the maximum size of payload will be tripled as shown in equation 5.

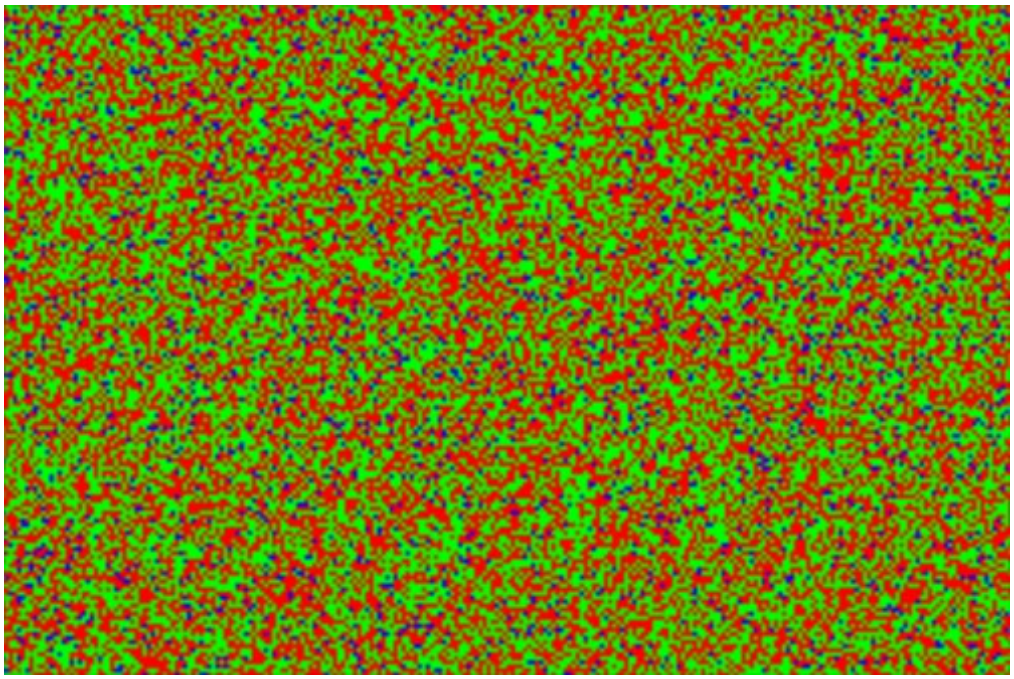

Figure 2: The (zoomed part of the) difference map of both images of figure 1. For perceptible visualization the difference values have to be mapped from range 0-1 to range 0-255. The map shows the **modification in every pixel** (in one channel), even a pixel does not contain part of the payload or the payload is stored in another channel with a none-modified channel value.

$$\begin{aligned} max_{(width*height)} &= width * height * cpp/8 - 4 \\ max_{(12MP)} &= 4256 * 2848px * 3/8 - 4 \\ &\widehat{=} 4545404\ B \widehat{=} 4,33\ MB \end{aligned} \tag{5}$$

Thus the storing is not done by the smallest possible modification anymore. However an intial and the resulting steganography image still look obviously identical.

5 Summary And Future Work

This paper presented a novel steganography approach, which firstly encrypts and secondly stores the payload directly within the image content by the smallest possible modification of a pixel. Encryption and choice of pixel are determinated highly dynamic constrained by one-way functions and hash functions. The deviation of all pixel compared to their initial values is $\delta_{RGB} = 0,1302\%$ and ensures that original and resultant image look virtual identically without any visible or qualitative decreases and independently of color depth.
LeonSteg deals directly with the image content (pixel-based). Hence a lossy compression method like JPEG cannot be used. It is needed to apply lossless compression like PNG. In future work it would be desirable to combine LeonSteg with a lossy compression method.

6 Acknowledgments

Special thanks to my son, who is responsible that I developed LeonSteg. As baby he could not fall asleep caused of aches and pains. Only by walking by baby carriage he could drift off. While walking with him and his baby carriage many hours through the 2nd garden cemetery of Berlin-Schöneberg I developed the approach of LeonSteg. Thanks a lot my son.
Furthermore special thanks to Dr. K. Krause for her big support in checking english grammar.

References

[1] Syed Ali Khayamb Hassan Khana, Mobin Javedb and Fauzan Mirzab. Designing a cluster-based covert channel to evade disk investigation and forensics. *Computers & Security*, 30(1):35–49, 2011.

[2] Neil F. Johnson. Steganography. *Technical Report*, November 1995.

[3] Neil F. Johnson and Sushil Jajodia. Steganography: Seeing the unseen. *IEEE Computer*, pages 26–34, 1998.

[4] Costas S. Iliopoulos Moudhi M. Aljamea and M. Samiruzzaman. Detection of url in image steganography. *Proceedings of the International Conference on Internet of things and Cloud Computing (ICC '16)*, 2016.

[5] Frank Nagl and Paul Grimm. StegHash - A Hash-based Approach For Image-Based Steganography. *Proceedings of the 22nd workshop of color image processing 2016*, pages 133–142, ISBN 978-3-00-053918-3, 2016.

[6] Zoran Duric Neil F. Johnson and Sushil Jajodia. *Information Hiding: Steganography and Watermarking - Attacks and Countermeasures*. Springer Science+Business Media, New York, NY, USA, 2000.

[7] Dongming Peng Prabhat Dahal and Hamid Sharif. Image processing pipeline model integrating steganographic algorithms for mobile cameras. *Proceedings of the 2016 ACM International on Workshop on Traffic Measurements for Cybersecurity (WTMC '16)*, pages 50–57, 2016.

[8] Heise Security. Tarnkappe kontra Krypto-Verbot: Computergestützte Steganographie versteckt Daten in Bild- und Audiodateien. https://www.heise.de/security/dienste/Tarnkappe-kontra-Krypto-Verbot-475025.html, 2010.

[9] Troonie. A portable tool to convert, trim, stitch, filter photos and work with steganography. http://troonie.com, 2017.

[10] Wikipedia. Steganography. https://en.wikipedia.org/wiki/steganography, 2017.