- StegHash -

A Hash-based Approach For Image-Based Steganography

Note: This is the white-paper version. The original paper was published in proceedings of 22nd workshop of color image processing, Ilmenau 2016, pages 133-142, ISBN 978-3-00-053918-3.

Figure 1: Initial image (left) with 2078x2770 px was processed by StegHash. The resulting image (middle) looks virtual identically. The complete GPL 3.0 license text [11] (right, excerpt) was encrypted, consisting of 674 lines, 5698 words and 35140 characters (including spaces).

as

ABSTRACT

This paper defines a novel hash-based approach for image-based steganography to encrypt longer text dynamically. ASCII characters will be encrypted by subtracting hash values of their byte values and storing the result into several RGB channels of several pixel. A constantly changing order of compromised pixel is ensured on the basis of dynamic dependencies of computing the distance between pixel to use. The dependencies comprise text length, key (password) used as input data for computing hashes and image resolution. Just one of the three RGB channels of a compromised pixel is used for encrypting. Determining the relevant RGB channel is done by modulo-3-check of the corresponding indexed hash value. This ensures dynamic use of RGB channels. The detection of compromised pixel is highly improbable. Assuming that a attacker could identify compromised pixel, the pixel solely contains encrypted values. Only after correct Hash-based decrypting the encrypted values results in the original plain text characters. By using a secured hash function, the decryption without the relevant key is impossible.

Keywords

steganography, encryption, hash function, image processing, steganalysis, cryptography, cryptology, decryption, troonie

1. INTRODUCTION

The origin of steganography dates back several centuries. The first recorded use of the term was in 1499 by Johannes Trithemius in his *Steganographia*, a treatise on cryptography and steganography, disguised as a book on magic [1]. But with the origination of digital photography and digital image processing, technically more mature approaches and tools for image-based steganography were published.

In contrast to common cryptography the steganography has the advantage that an intended secret message does not attract attention to itself as an object of scrutiny [1]. A disadvantage constitutes the possibility of decryption of the secret message, when relevant pixel could be identified and their value could be interpreted correctly.

On this basis, the approach of StegHash combines steganography and conventional encryption in a dynamic way. The StegHash algorithm depends on the result of a secured hash function, a key (password) used as input data for the hash function, the text length and the image resolution. This ensures a dynamic steganographic mechanism (constantly changing order of compromised pixel) as well as a secured encryption of the plain text.

2. STATE-OF-THE-ART

The research area of image-based Steganography produced many publications and tools in past. Already in 1995 and 1998 Neil F. Johnson published technical reports to give an overview about the history of (image-based) Steganography [2][3]. Furthermore, digital data hiding techniques for images were explored, analyzed, attacked and countered [4]. The website Heise Security introduced and evaluated in 2010 eighteen different Steganography tools [5]. Also newer publications describe novel and inventive approaches and application areas for Steganography [6] [7][8]. In 2011, a steganograhic mechanism by selective hard disc fragmentation was depicted [6]. Latest publications focus on mobile cameras as goal for steganographic algorithms [7] and explain the detection of URLs in image-based Steganography [8].

In contrast to all other publications and tools the novel approach of this paper uses conventional encryption to construct the steganographic mechanism.

3. CONCEPT

StegHash is a hash-based algorithm to encrypt longer text into color images without visible or qualitative decreases of the image content.

3.1 Basic idea

The basic idea of StegHash is to **store digit numbers into the units of one of the RGB channels of a pixel whereas the tenner and the hundreds of the channel value will be preserved**. Previously the byte value of an ASCII character was fractionalized into hundreds, tens and units, all as digit numbers.

Figure 2 illustrates the basic idea using three pixel for one ASCII character. The byte value of ASCII character 'a' is 097. This number is fractionalized into 0 (hundred), 9 (tenner) and 7 (unit). In the next step the three digit numbers are stored in three different pixels (P1, P2, P3) in another color channel, respectively. As a result the three modified pixels (P1', P2', P3') encrypt the byte value of an ASCII character whereas the color value of the pixel is changed only marginal.

Accordingly all ASCII characters of a text string will be split in their three digit numbers and stored into a byte triple array, called `charTriple`.

```
numberUsablePx =
width * height - sumHashElements – width;

startH = sumHashElements / width;

startW = sumHashElements - width * startH;

distance =
numberUsablePx / charTriple.Count – 1;

encryptValueForDistance =
GetDigitSumOfByte(hash[digitSumOfHashElements]);

encryptedDistance =
distance - encryptValueForDistance;
```

Code snippet 1: The computation of initial values of StegHash as pseudo code (in C# style).

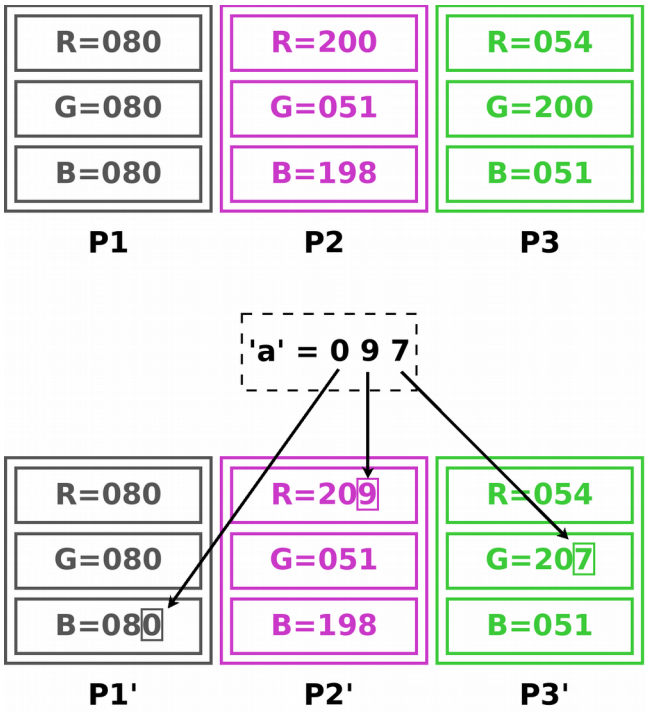

Figure 2: Basic idea of StegHash. The three digit numbers of the byte value of character 'a' are stored in three different pixels (P1, P2, P3) in another color channel, respectively.

3.2 Start position and distance between encrypting pixels

By using StegHash a `key` (password) is necessary. The `key` is used by a secured hash function (designed as a one-way function) to map the `key` to a bit string of fixed size (called `hash` byte array). In the implementation of StegHash two hash functions (SHA512 and MumurHash) are combined (*section 4*). Essentially every secured (single) hash function can be used.

A constantly changing order of compromised pixel is ensured on the basis of dynamic dependencies for computing the `distance` between the pixels to use. The dependencies are the text length (`charTriple.Count`), the `key` for computing the `hash`es and the image resolution (`width`, `height`).

The number `sumHashElements` is the sum of all hash array elements added together. It is used as starting pixel position by linear scanning for initiating the StegHash algorithm. Resulting from image resolution and `sumHashElements` a usable number of pixels for encryption is computed (called `numberUsablePx`). This value as well as the text length is used for computing the `distance`.

The digit sum of `sumHashElements` (called `digitSumOfHash-Elements`) points (as index) to an element of the `hash` array.

The additional summing also the digits of this element results in a number (called `encryptValueForDistance`) used for subtraction from distance as `encryptedDistance`. *Code snippet 1* shows the computation (C# style).

3.3 Encryption by hash values

Expanding to the basic idea (*section 3.1*) the split three digit numbers of ASCII characters of the plain text will not be stored directly in `charTriple`. In fact digit numbers will be encrypted by subtracting `hash` values, analogous to the encryption of `encryptedDistance` (*section 3.2*). *Code snippet 2* shows the encryption.

```
int index = 0;

foreach (byte item in textInByte) {

   int one, ten, hundred;

   byte encryptedItem = item - hash[index]);

   Fractionalize(encryptedItem, out hundred,
   out ten, out one);

   charTriple.Add (hundred);

   charTriple.Add (ten);

   charTriple.Add (one);

   index++;

}
```

Code snippet 2: Encrypting plain text (as bytes) by subtracting hash values.

3.4 Storing encrypted values into pixels

Before storing any ASCII content the `distance` respectively the `encryptedDistance` has to be saved. By interpreting an image as an 2-dimensional array and by linear scanning the StegHash algorithm starts its encryption of the `encryptedDistance`.

The number `sumHashElements` points (as index) to the pixel of origin. This and the next three pixel will be used to store the fractionalized `encryptedDistance`.

Starting exactly one line below the pixel of origin, StegHash jumps to all relevant encrypting pixels by the given `distance`. An encrypting pixel will be split in its RGB channel values. Exclusively one of the three channel values will be modified at its unit value with the passed `charTriple` array element (as described in *section 3.1*). The determination of the RGB channel that has to be modified is done by modulo-3-check of the indexed hash array element. **This ensures different utilization of RGB channels**.

Code snippet 3 and *figure 3* show an example to demonstrate the principle of storing values into pixels.

3.5 Decryption

Analogous to encrypting process the `encryptedDistance` has to be readout and decrypted with the `encryptValueForDistance` to obtain the correct `distance` value. The `sumHashElements` value is used to point (as index) to the pixel of origin. Again, starting from pixel of origin the `encryptValueForDistance` is readout. Starting exactly one line below the pixel of origin, StegHash jumps to all relevant encrypting pixel by the given `distance`, determines the RGB channel, extracts its unit value and decrypts the byte value before adding it to the byte triple array `charTriple`. By defractionalizing the byte triple the original character (as byte value) will be decrypted (by summing `hash` values).

```
width = 10

height = 10

hash = [3, 6, 1, 2]

sumHashElements = 12

numberUsablePx = 100 – 12 = 88

distance = 88 / 18 ≈ 4 → 0004

text = Hello! =
['H', 'e', 'l', 'l', 'o', '!']

textInByte =
[072, 101, 108, 108, 111, 033]

charTriple =
[0,7,2,1,0,1,1,0,8,1,0,8,1,1,1,0,3,3]
```

Code snippet 3: Example for storing the text 'Hello!' into an image with the resolution 10x10 px. For better illustration the `distance` and the ASCII content (`charTriple`) are not encrypted.

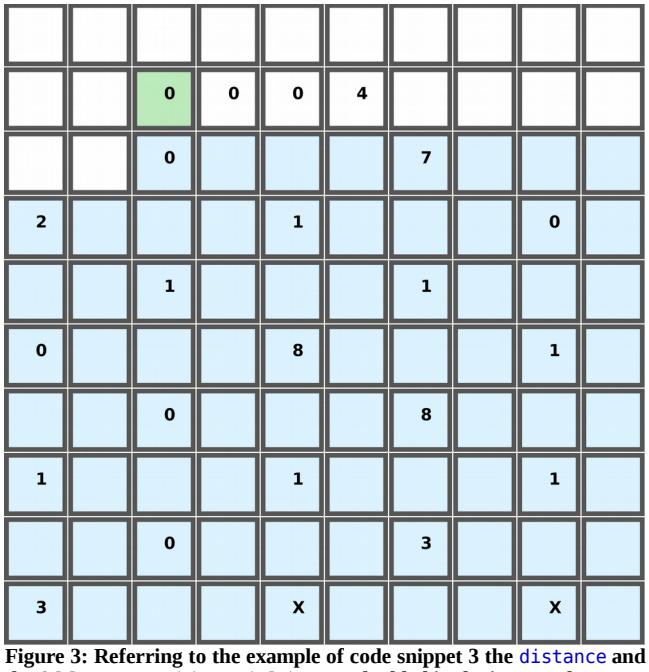

Figure 3: Referring to the example of code snippet 3 the `distance` and the ASCII content (`charTriple`) are embedded in the image. The green highlighted pixel is the pixel of origin and all light-blue highlighted pixels represent the usable pixel area for storing `charTriple`.

4. IMPLEMENTATION DETAILS

The implementation of StegHash is realized within the software Troonie [9]. Troonie and StegHash are realized by using Mono/.NET [10]. The code snippets in this paper are taken directly from the implementation and hence in C# style.
The concept is virtual exactly implemented as described but with the addition of four enhancements:

a) It is highly improbable to identify compromised pixels. However, the implementation provides an (optional) obfuscation mechanism for prevention and avoidance of a possible detection.

Before the StegHash algorithm starts, the image will be preprocessed. Similar to the basic idea of StegHash (see *section 3.1*), the units of one of the RGB channels (channel is determined randomly) of all pixels will be manipulate by pseudo-random values whereas the tenner and the hundreds of the channel value will be preserved. This generates a kind of pseudo-noise and compromised pixels cannot be distinguished from other pixels. By implementing the boolean switch `UseStrongObfuscation` this optional obfuscation mechanism will be (de-)activated.

b) Before working with the ASCII plain text to encrypt by StegHash, a value called `asciiMoveValue` is subtracted from the ASCII byte values. `asciiMoveValue` is computed by the digit sum of the length value of the `key` (password). This ensures that the original plain text is not used directly in StegHash algoritm, but a (of course not-safe) pre-encrypted text.

c) Like described in *3.2* the `key` is used by a secured hash algorithm to map it to a bit string of a fixed size (called `hash` byte array). In the implementation of StegHash two hash algorithms (SHA512 and MumurHash) are combined. By using simple one-way functions the original `key` is used to compute two separate `key`s. This ensures that every hash algorithm uses separate mapping data.

d) In implementation a special character is diverted from its intended use. The character '~' (byte value 126) is used to identify a line break. It will be added at the end of every line in the plain text. Consequentially a text containing this character will be misinterpreted. At the position of the character a line break will be inserted.

5. RESULTS

The deviation of the value of a compromised pixel from its original value (in exclusively one of the three color channels) value amounts **maximum 9 / (3 * 256) = 1.172 % and on average 0.586 %**, in which 9 is the maximum possible difference between two unit values and 3 * 256 represents the RGB byte channels of the pixel. The dynamic steganographic mechanism results in a constantly changing order of the compromised pixels. It is highly difficult to detect a possible deviation of a compromised pixel. In *figure 1* the initial image with 2078x2770 px results in an image, **which looks virtual identically**. The complete GPL 3.0 license text [11] was encrypted, consisting of 674 lines, 5698 words and 35140 characters (including spaces).
A second example is shown in *figures 4* to *9*. The initial image in *figure 5* with 400x250 px was used to encrypt the text, shown in *figure 4*.

```
Jingle bells, jingle bells,
jingle all the way
O, what fun it is to ride in a
one-horse open sleigh
```

Figure 4: Second example. Used plain text to encrypt.

Again, the resulting image (*figure 6*) looks virtual identically. In *figure 7* the difference image is shown. For better visualization the difference values are mapped from range 0-9 to range 0-255.

Figure 5: Initial image of second example.

Figure 6: Resulting image, looking virtual identically to initial image.

To identify compromised pixels, an attacker

- requires the initial and the resulting image, or
- is able to compute `distance` as well as `sumHashElements` value, or
- uses a well-done steganalysis mechanism, which could detect these pixels.

All three options are highly improbable.

In *section 4a* an optional obfuscation mechanism for prevention and avoidance of a possible detection is explained. Considerably more image content is manipulated, since every pixel will be modified. Nevertheless the identification of compromised pixels is again more difficult and more improbable.

In *figure 8* the difference image by using the optional obfuscation mechanism is shown. *Figure 9* shows the comparison of a zoomed image section. Again, the initial (left), the resulting image without (middle) and the resulting image with optional obfuscation mechanism (right) look virtual identically. Only in bigger monochromatic areas a slight visual difference between initial and resulting image with optional obfuscation mechanism is noticeable.

Assuming an attacker could identify compromised pixels, the pixels contain encrypted values (*figure 10*). Only after the correct hash-based decryption the encrypted values result in the original plain text characters. Under the condition that a secured hash function is used, **the decryption without the relevant key is infeasible**.

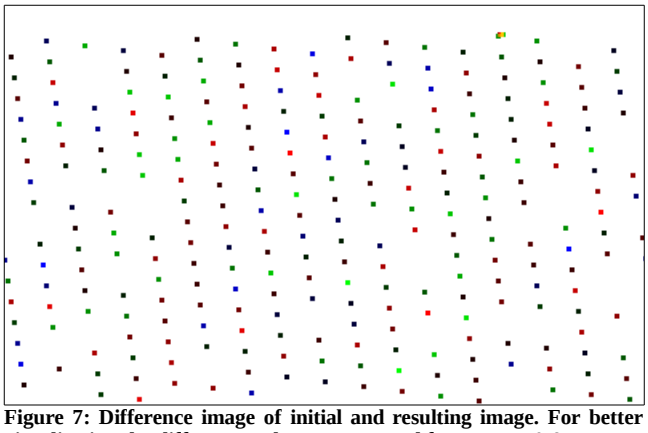

Figure 7: Difference image of initial and resulting image. For better visualization the difference values are mapped from range 0-9 to range 0-255.

Figure 8: Difference image by using the optional obfuscation mechanism (described in *section 4a*). For better visualization the difference values are mapped from range 0-9 to range 0-255.

Figure 9: Comparison of a zoomed image section. Again, the initial (left), the resulting image without (middle) and the resulting image with optional obfuscation mechanism (right) look virtual identically. Only in bigger monochromatic areas a slight visual difference between initial and resulting image with optional obfuscation mechanism is noticeable.

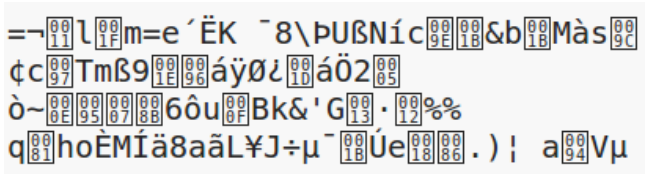

Figure 10: Not decrypted values of the compromised pixels. Values represents the (encrypted) text given in figure 4.

6. SUMMARY AND FUTURE WORK

6.1 Summary

In this paper StegHash, a novel hash-based approach for image-based steganography to encrypt longer texts dynamically, is presented. For this purpose ASCII characters of the plain text are encrypted and stored into image pixels by a dynamic mechanism.

The encryption as well as the decision, which pixel are used by StegHash depends on the result of a secured hash function, a key (password) used as input data for the hash function, the text length and the image resolution. This ensures a constantly changing order of the compromised pixels.

The byte values of the ASCII characters are fractionalized into hundreds, tens and units, all as digit numbers. These digit numbers are stored into the units of one of the RGB channels of a pixel whereas the tenner and the hundreds of the channel value remain unchanged.

The determination of the RGB channel that has to be modified is done by modulo-3-check of the indexed hash array element. This ensures different utilization of RGB channels.

The difference between an origin and a compromised pixel amounts maximum 1.172 % and on average 0.586 %. This ensures that original and resulting image look virtual identically.

The encryption of the plain text by usage of a secured hash function before storing the text inside the image ensures that a highly improbable detection of compromised pixel would be useless.

6.2 FUTURE WORK

The current approach of StegHash focuses on the usage of longer texts. An enhancement could extend the usage not only for text but also for digital photos or other digital media files. Therfor, photos or other media files should be fragmented into byte arrays. Essentially every byte array could be used by StegHash.

In this implementation a static initial image is needed and also results in a static image. The usage of video streams is also imaginable. This would increase the capacity of text that might be encrypted enormously. Hence, a complete book could be encrypted in a short (lossless compressed) video.

StegHash works directly with the image content and changes pixel values in a very small range. Hence the saving process of a steganographic image needs a lossless data compression like the PNG image format [12]. Future work could provide the usage of lossy compression like the famous image format JPEG [13].

7. ACKNOWLEDGMENTS

Sincere thanks to our families and friends for supporting our work every day. Also special thanks to the Mono community for developing the great Mono Project [10]. Furthermore special thanks to Dr. K. K. for her big support in checking grammar and linguistic usage of this paper.

8. REFERENCES

[1] Wikimedia Foundation, Wikipedia-Artikel. Steganography. https://en.wikipedia.org/wiki/Steganography. 2016.

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

[3] Neil F. Johnson and Sushil Jajodia. Steganography: Seeing the Unseen. In IEEE Computer: 26-34. 1998.

[4] Neil F. Johnson, Zoran Duric and Sushil Jajodia. Information Hiding: Steganography and Watermarking - Attacks and Countermeasures. 2000.

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

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

[7] Prabhat Dahal, Dongming Peng, and Hamid Sharif. Image Processing Pipeline Model Integrating Steganographic Algorithms for Mobile Cameras. In Proceedings of the 2016 ACM International on Workshop on Traffic Measurements for Cybersecurity (WTMC '16): 50-57. ACM. 2016.

[8] Moudhi M. Aljamea, Costas S. Iliopoulos, and M. Samiruzzaman. Detection Of URL In Image Steganography. In Proceedings of the International Conference on Internet of things and Cloud Computing (ICC '16), Article 23. ACM. 2016.

[9] Troonie Project. Troonie - A portable tool to convert, trim, stitch, filter photos and work with steganography. http://www.troonie.com. 2016.

[10] Xamarin. Mono Project. http://www.mono-project.com. 2016.

[11] Free Software Foundation, Inc. GNU General Public License. Version 3. https://www.gnu.org/licenses/gpl-3.0.html. 2007.

[12] International Organization for Standardization (ISO). ISO/IEC 15948:2004. Information technology -- Computer graphics and image processing -- Portable Network Graphics (PNG): Functional specification. http://www.iso.org/iso/-catalogue_detail.htm?csnumber=29581. 2004.

[13] Joint Photographic Experts Group. Overview of JPEG. https://jpeg.org/jpeg/. 1992/2014.