

6.857 Final Project: Image Steganography

Dean Fanggohans

Caroline Jin

May 10, 2022

Abstract

As people use multimedia like images, audio, and video on the internet, it is important that this data is securely sent to the recipient. Encryption can prevent an adversary from learning about any content in the media, but the attacker knows that sensitive data is transmitted between two users. Image steganography enables the user to conceal the existence of a secret message by hiding in some cover images. In this paper, we will explore different image steganography methods and analyze the security of each technique.

1 Introduction

Steganography enables invisible communication between the sender and receiver by hiding a secret inside some cover medium. It has been used since the age of the Greeks when Histaeus sent secrets by tattooing a slave's scalp, waiting for the hair to grow and cover the scalp, and then sending the slave to deliver the secret. Even during World War II, the Germans used null ciphers where for instance, the secret was every third letter in a given message [7, 5].

In the digital world today, steganography methods are divided into categories based on different cover mediums like image, audio recording, video, or even text. People often use images due to its high capacity to hold data. Images also contain redundant data, so changing a few bits for the secret does not mutate the image as much—especially the least significant bits of the image data.

Broadly speaking, image steganography can be further divided into two groups based on the domain it is being performed: spatial and frequency. With spatial domain, the secret is embedded directly by mutating the pixel values of the cover image, while the frequency domain instead involves embedding the secret in the frequency representation of the cover image, which can be obtained through some transformation [5, 6].

In this paper, we will be analyzing a method called LSB for the spatial domain and another method called DCT-LSB (which we will refer to as just DCT, for brevity) for the frequency domain. We will evaluate these two methods based on how well the secret can be recovered from the cover image in Section 3 and the security of these two methods in Section 4. We will also discuss how adding key into these steganographic method can enhance our security guarantees in Section 4.2.

2 Implementation

2.1 LSB

We represent an image as a three-dimensional array of 8-bit unsigned integers, where the dimensions correspond to height, width, and channel (RGB images will have 3 channels, for instance). To embed the secret data, the least significant bits (LSB) of the image are flipped to the bit-value of the secret data [6]. For instance, let's say we are embedding secret '110' in an RGB image. Since each pixel value has 3 channels, we can embed all three bits of this secret using just one LSB. The new pixel value would look like as follows:

$$(10001101,00110101,11101100)$$

The idea for LSB is that changing the LSB should not add too much noise into the cover image. We discuss later the potential security flaws through statistical analysis in Section 4.1.2.

2.1.1 Code

```

1 def embed(cover_mat: IntArr, data: bytes, key: bytes = None, num_lsb: int = 1) -> IntArr:
2     flat_cover = cover_mat.flatten()
3
4     # Perform random permutation with RNG seeded by the key
5     if key:
6         perm, inv_perm = _generate_permutation(key, len(flat_cover))
7         flat_cover = flat_cover[perm]
8
9     # Check hiding capacity
10    if 8*len(data) > len(flat_cover) * num_lsb:
11        raise ValueError('data exceeded the hiding capacity')
12
13    # NOTE: Match the dtype here (otherwise there would be a weird casting bug)
14    data_bitarray = _split_from_bytes(data, num_lsb).astype(flat_cover.dtype)
15
16    flat_cover[:len(data_bitarray)] &= ~((np.ones_like(data_bitarray) << num_lsb) - 1) #
17    Remove 'num_lsb' LSBs
18    flat_cover[:len(data_bitarray)] |= data_bitarray # Write data into LSBs
19
20    # Invert permutation
21    if key:
22        flat_cover = flat_cover[inv_perm]
23
24    stego_mat = np.reshape(flat_cover, cover_mat.shape)
25    return stego_mat
26
27 def extract(stego_mat: IntArr, key: bytes = None, num_lsb: int = 1) -> bytes:
28     flat_stego = stego_mat.flatten()
29
30     if key:
31         perm, _ = _generate_permutation(key, len(flat_stego))
32         flat_stego = flat_stego[perm]
33
34     return _merge_to_bytes(flat_stego, num_lsb)

```

Listing 1: LSB embedding and extraction

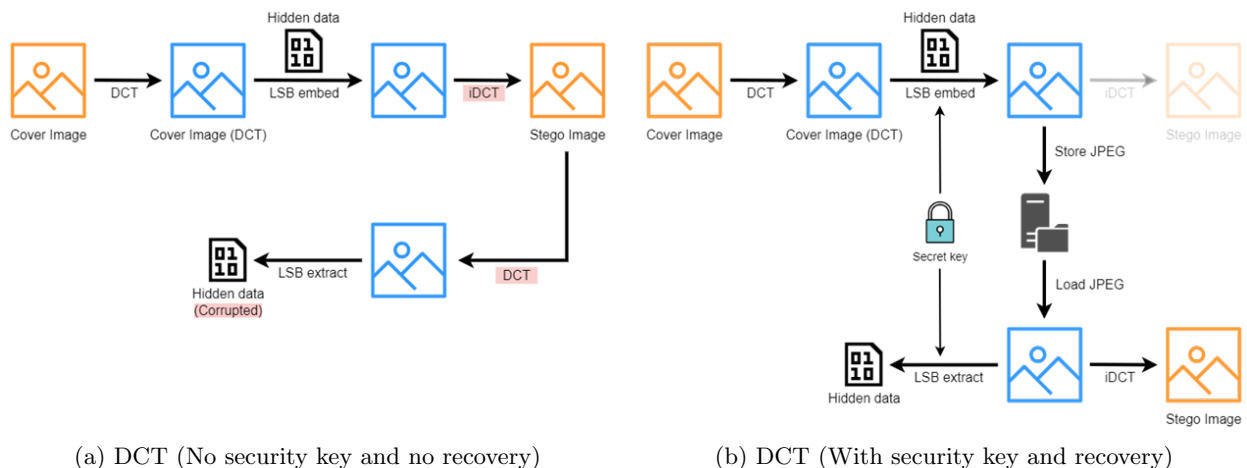


Figure 1: DCT diagram comparison between naive (i.e. no security and recover information) and fixed (i.e. security key and recovery)

2.2 DCT

In DCT, the process starts with converting the cover image into its frequency domain using DCT (Discrete Cosine Transform), and then proceeds to using LSB method to embed the data. To obtain the stego image (i.e. cover image with the embedded secret), we convert this embedded DCT image back into its spatial domain using inverse DCT. This process is shown in Figure 1a.

One subtlety that arises from this method is that DCT (and its inverse) produces real numbers rather than integer values. This introduces some rounding, which corrupts the hidden data embedded in the frequency domain. While this is not an issue for generating the stego image, we will not be able to recover our secret back if the stego image is not stored properly (as shown in Figure 1a).

In order not to corrupt the embedded secret, we need to keep the embedded DCT image in its frequency domain while storing it on disk. In fact, this scheme is done by some image formats like JPEG (as long as we don't additionally perform lossy compression on the DCT values). For instance, as long as we store the DCT image in the correct JPEG format, any standard image viewer will be able to generate the stego image (i.e. in spatial domain) for us. Then, we can perfectly recover the secret back by reading the LSB values from the DCT image, since no additional DCT or inverse DCT is performed between the LSB embedding and recovery (as shown in Figure 1b).

2.2.1 Code

```
1 def dct(mat: ImageMatrix) -> DctOutputMatrix:
2     nrows, ncols = mat.shape[0], mat.shape[1]
3     nchannels = 1 if len(mat.shape) == 2 else mat.shape[2]
4
5     if nrows % BLOCK_SIZE != 0 or ncols % BLOCK_SIZE != 0:
6         raise ValueError(f'Width and height of the image must be multiple of {BLOCK_SIZE}')
7
8     # Temporarily expand grayscale matrix into 1-channel image
9     if len(mat.shape) == 2:
10        mat = np.expand_dims(mat, axis=2)
11
12    # Recenter [0, 255] -> [-128, 127]
13    signed_mat = mat.astype(np.int16) - 128
14
15    # Perform DCT block-wise
16    dct_mat = np.zeros_like(signed_mat, dtype=np.int32)
17    for i in range(0, nrows, BLOCK_SIZE):
18        for j in range(0, ncols, BLOCK_SIZE):
19            for k in range(nchannels):
20                block = signed_mat[i:i+BLOCK_SIZE, j:j+BLOCK_SIZE, k]
21                dct_mat[i:i+BLOCK_SIZE, j:j+BLOCK_SIZE, k] = _dct(block)
22
23    # Revert matrix expansion
24    if len(dct_mat[0][0]) == 1:
25        dct_mat = np.squeeze(dct_mat, axis=2)
26
27    return dct_mat
28
29 def idct(dct_mat: DctOutputMatrix) -> ImageMatrix:
30     nrows, ncols = dct_mat.shape[0], dct_mat.shape[1]
31     nchannels = 1 if len(dct_mat.shape) == 2 else dct_mat.shape[2]
32
33     if nrows % BLOCK_SIZE != 0 or ncols % BLOCK_SIZE != 0:
34         raise ValueError(f'Width and height of the image must be multiple of {BLOCK_SIZE}')
35
36     # Temporarily expand grayscale matrix into 1-channel image
37     if len(dct_mat.shape) == 2:
38        dct_mat = np.expand_dims(dct_mat, axis=2)
39
40    # Perform IDCT block-wise
41    idct_mat = np.zeros_like(dct_mat, dtype=np.int16)
42    for i in range(0, nrows, BLOCK_SIZE):
43        for j in range(0, ncols, BLOCK_SIZE):
```

```

44         for k in range(nchannels):
45             block = dct_mat[i:i+BLOCK_SIZE, j:j+BLOCK_SIZE, k]
46             idct_mat[i:i+BLOCK_SIZE, j:j+BLOCK_SIZE, k] = _idct(block)
47
48         # Revert matrix expansion
49         if len(idct_mat[0][0]) == 1:
50             idct_mat = np.squeeze(idct_mat, axis=2)
51
52         # Floor and ceil out-of-bound values
53         idct_mat = np.minimum(np.maximum(idct_mat, -128), 127)
54
55         return (idct_mat + 128).astype(np.uint8) # Recenter [-128, 127] -> [0, 255]

```

Listing 2: DCT embedding and extraction

2.3 Key-Based Steganography

We can further enhance both the LSB and DCT methods with additional layer of security: Even if an adversary is able to detect the presence of a hidden secret inside of stego image, they should not be able to extract the secret—a property we refer to as unextractability (see our discussion in Section 4).

We achieve this in our implementation by randomly permuting the entire pixels of the cover image before we embed the secret. Note that in the previous methods, we simply fill in the LSB linearly (i.e. from top-left to bottom-right), but by using this method all bits of the secret data will effectively be randomly scattered throughout the cover image. To recover the secret, we need to be able to invert the permutation deterministically, in which case we can use a source of randomness (basically a PRF) seeded with a symmetric key shared between the sender and the recipient. This will allow the recipient to invert the random permutation as long as they know the secret key. On the other hand, an adversary will not be able to generate this invert permutation, and thus unable to determine the correct ordering of the secret embedded in the LSB.

2.4 Experiments

We experimented with using three different cover images shown in Table 1 of size 512x384 pixels and embedding the same secret image of size 128x96 pixels, which is a quarter of the size of the cover image. For our experiments, we worked on varying the steganography methods, whether a key should be used, and how many LSBs per pixels we should embed the image.

3 Recovered Image

Before analyzing security, we wanted to evaluate whether the recipient can extract the message from the stego image. The results of the percentage of the secret image recovered is shown in Table 1. With LSB, the recipient can fully recover the secret whereas in DCT (using the naive approach), due to rounding error, the recipient can only partially recover the image. Due to this partial recovery, encrypting the secret and then embedding DCT is not recommended.

4 Security

When evaluating for security, we considered the two following security guarantees [4]:

1. "Imperceptibility": The cover image should not degrade after applying the steganography method. The secret should not be detected.
2. "Unextractability": The adversary cannot extract any part of the hidden secret from the steganographic image.

These security guarantees are layered. If the method offers the first security guarantee, the second security follows through as an adversary without knowing a secret exists in the first place will not attempt to extract. However, if the adversary does know the secret exists but the method provides the second guarantee, then at least secret remains unknown.










Cover Image	Recovered Image (LSB)	Recovered Image (DCT naive)
	 100.0	 57.33
	 100.0	 59.76
	 100.0	 45.91

Table 1: This table shows a comparison of the recovered images from different steganography methods as well as the percentage of pixels recovered.

4.1 Imperceptibility

In evaluating the first guarantee, we decided to use the following criteria: distortion measure with PSNR and statistical analysis with RS-steganalysis.

Distortion measures involve considering the difference between the stego image (i.e. cover image containing the secret) and original cover image. PSNR is a common distortion measure used for the security of image steganography [6, 4], given that the adversary only has the ability to distinguish between the two images.

However, this is not the case as the adversary has the power to repeatedly ask some encoding oracle to embed the hidden data for a given cover image. Different amounts of embedding can help the adversary learn how a stego image differs from cover image. This is the approach for RS-steganalysis [1]. For our analysis, we will be using both PSNR and RS-steganalysis.

4.1.1 PSNR

Peak signal-to-noise ratio (PSNR) measures how distorted the stego image is compared to the base cover image. The higher the PSNR value, the less the distortion, meaning the secret from a visual perspective is embedded well in the cover image. A value above 30 dB means is a threshold for an acceptable distortion value [6], but any PSNR value less than that means the distortion is too high and can be detected. The results of embedding a secret image into different covers images using different steganography methods are shown for in Table 2. In general, both methods have comparable PSNR values across different cover images.

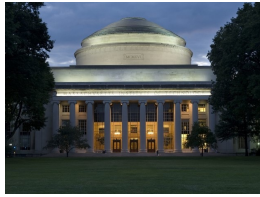
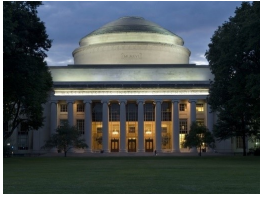
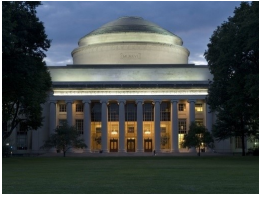






Cover Image	Stego Image (LSB)	Stego Image (DCT)
	 46.43	 46.20
	 39.91	 39.84
	 44.27	 44.07

Table 2: This table shows a comparison of the original cover images (512x384 pixels) and stego images (512x384 pixels) resulting from different steganography methods as well as the PSNR value. For both methods, we are storing the hidden image (128x96 pixels) in only one LSB per pixel without using a key.

4.1.2 RS-Steganalysis

While using LSB has made visual analysis through distortion metrics like PSNR difficult, Fridrich et al. [1, 3] found that there was a strong correlation between bit planes using LSB steganography. Using this fact, they develop RS-steganalysis, which can detect the presence of a secret and how much of the secret is embedded in the cover image.

RS steganalysis involves the following procedure [1, 3]. An image is divided into groups of neighboring pixels; a discrimination function is applied to each group to determine its noisiness (i.e. higher noise is equivalent to higher difference between adjacent pixels). Noise is injected into each group using one of two flipping functions F_1 and F_{-1} . F_1 flips the least significant bit like LSB would where 1 changes to 0, 0 to 1, 2 to 3, 3 to 2, and so on. F_{-1} changes pixel values similarly from -1 to 0, 0 to -1, 1 to 2, and so on. After applying the flipping function, the discrimination function is applied to noise-injected group and the result is compared to the original group. Groups are divided into R -group (increase in noise), S -group (decrease in noise), and neutral (no change).

For images with no embedding, the difference between the R and S groups should be approximately the same regardless of the flipping function used. However, this is not the case for LSB-embedded image, in which the ratio of R groups will be greater when applying F_{-1} vs when applying F_1 . The difference between R and S groups when applying F_{-1} increases and decreases for F_1 when embedding more secret data into the cover image. The comparison of this difference is shown in Figure 2. We see that as we embed more data for LSB their is a distinctive increase in difference for F_{-1} and decrease in different for F_1 whereas DCT remains fairly constant.

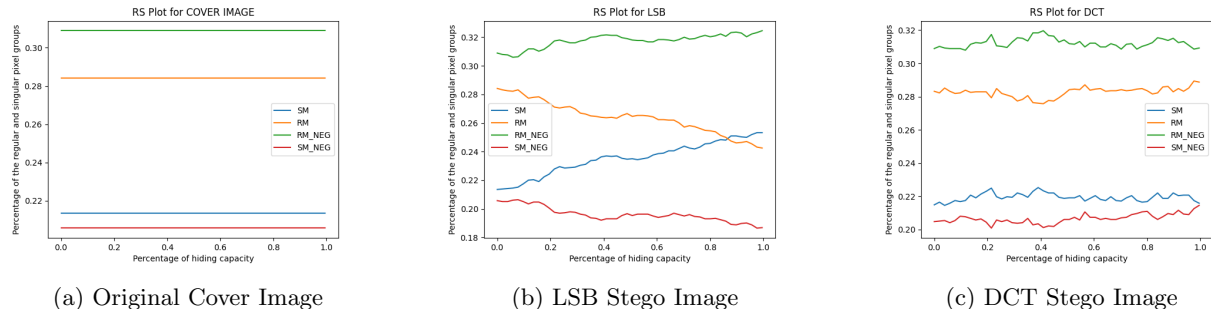


Figure 2: RS plot comparison between original, LSB, and DCT.

4.1.3 Other Statistical Analysis

RS-steganalysis attacks focus primarily on revealing LSB steganography. There are also other steganalysis attacks for DCT, such as using the properties of AC coefficients [2]. There still work that needs to be done on defining concrete metrics of measuring imperceptibility.

4.2 Unextractability

The adversary should not be able to extract any part of the hidden secret from the stego image. One way to achieve this is by combining cryptography, i.e. encrypting the secret before we embed it, but in this paper we try to look more into some key-based steganography techniques.

Without any additional technique, both LSB and DCT can be vulnerable to extraction once the adversary knows that there exists a secret. However, using our method of randomly permuting the pixels of the embedded image, no one except the sender and the recipient can extract the hidden secret from the stego image, since it can only be done with the knowledge of the secret key. Furthermore, we also claim that this method is secure (in a similar sense to how we define security in encryption), which follows directly from the property of PRFs.

5 Future Work

While LSB (both in spatial and frequency domain) is the most common technique used to perform steganography, recent approaches tend to combine LSB with other techniques. For instance, an image steganography technique called F5 uses matrix encoding in order to achieve higher hiding capacity (i.e. the amount of data that can be embedded inside a cover image) [9]. Some other methods use various kinds of encoding and compression techniques, as well as other schemes for performing key-based steganography.

Some research can also be done on finding better evaluation metrics to evaluate the imperceptibility of image steganography techniques, e.g., more statistical-based metrics rather than just relying on the visual perception of human eyes. On the other spectrum, there are also some works that explore the possibility of having a public-key steganography [8], which will definitely be worth exploring.

6 References

References

- [1] Jessica Fridrich and Miroslav Goljan. Practical steganalysis of digital images: state of the art. *security and Watermarking of Multimedia Contents IV*, 4675:1–13, 2002.
- [2] Mao Jia-Fa, Niu Xin-Xin, Xiao Gang, Sheng Wei-Guo, and Zhang Na-Na. A steganalysis method in the dct domain. *Multimedia Tools and Applications*, 75(10):5999–6019, 2016.
- [3] SIMON JOHANSSON and EMIL LENNGREN. Steganographic embedding and steganalysisevaluation: An evaluation of common methods for steganographic embedding and analysis indigital images., 2014.
- [4] Inas Jawad Kadhim, Prashan Premaratne, Peter James Vial, and Brendan Halloran. Comprehensive survey of image steganography: Techniques, evaluations, and trends in future research. *Neurocomputing*, 335:299–326, 2019. URL: <https://www.sciencedirect.com/science/article/pii/S0925231218312591>, <https://doi.org/https://doi.org/10.1016/j.neucom.2018.06.075> doi:<https://doi.org/10.1016/j.neucom.2018.06.075>.
- [5] Tayana Morkel, Jan HP Eloff, and Martin S Olivier. An overview of image steganography. In *ISSA*, volume 1, pages 1–11, 2005.
- [6] Anita Pradhan, Aditya Kumar Sahu, Gandharba Swain, and K Raja Sekhar. Performance evaluation parameters of image steganography techniques. In *2016 International Conference on Research Advances in Integrated Navigation Systems (RAINS)*, pages 1–8. IEEE, 2016.
- [7] Alan Siper, Roger Farley, and Craig Lombardo. The rise of steganography. *Proceedings of student/faculty research day, CSIS, Pace University*, 2005.
- [8] Luis von Ahn and Nicholas J. Hopper. Public-key steganography. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 323–341, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [9] Andreas Westfeld. F5—a steganographic algorithm: High capacity despite better steganalysis. In *4th International Workshop on Information Hiding*, pages 289–302. Springer-Verlag, 2001.