International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 13 Issue: 11 | Nov 2026 www.irjet.net p-ISSN: 2395-0072

Least Significant Bit Steganography for Hiding Text and Images in Digital Photographs: A Python Implementation and Visual Evaluation

Toran V Athani[1]

[1]*Graduate (B.Tech, Computer Science and Engineering), Vellore Institute of Technology, Vellore, Tamil Nadu, India*

Abstract - *Steganography hides the existence of a message rather than its meaning, which makes it a useful companion to encryption where an obviously encrypted payload would itself attract attention. This paper presents two Python implementations of least significant bit (LSB) embedding and reports what was observed when they were run. The first hides an ASCII text message in the colour channels of a cover photograph using OpenCV, with a five-asterisk delimiter marking the end of the message. The second hides one image inside another using Pillow by keeping the high-order bits of the cover and storing the high-order bits of a resized secret image in the low-order positions. A 4-bit/4-bit split is compared with a 6-bit/2-bit split. The 13-character test message, which occupied 144 bits with its delimiter, produced no visible change in a cover image with a reported capacity of 94,357 bytes, and the decoder recovered it from the saved PNG file. In the image experiment, the 4-bit/4-bit merge produced a noisy recovered image, whereas the 6-bit/2-bit merge produced a recognisable but coarsely banded one. Analysis of the pipeline indicates that lossy JPEG storage in the first trial is the probable cause of the noise. The evaluation is visual only, and quantitative image-quality measures and steganalysis tests are identified as the next steps.*

Key Words: Steganography, Least Significant Bit, Image Steganography, Text Hiding, OpenCV, Pillow, Information Hiding

1. INTRODUCTION

1.1 Background

Cryptography protects what a message says. It does not hide the fact that a message exists. An intercepted ciphertext is still recognisable as ciphertext, and in some settings its presence alone draws scrutiny [1]. Steganography takes the opposite approach: the secret is placed inside an ordinary-looking cover object so that an observer has no reason to look for it [2]. The word comes from the Greek terms for covered and writing.

The two techniques are complementary. A sender can encrypt a message and then embed the ciphertext in a cover, so that an adversary has to detect the payload and also break the cipher [3]. The price of hiding is capacity. A cover can absorb only a limited number of changes before they become visible or statistically detectable [4], [5]. Reported uses include covert communication, digital watermarking for ownership claims [6], forensic analysis, and, on the adversarial side, concealment of malware in innocuous files [7].

Digital images are the most common cover medium, and LSB replacement is the most common embedding method because it is simple to implement and changes each channel value by at most one level [8], [9].

1.2 Related Work

LSB replacement is easy to detect when the payload is large. The chi-square attack [10], the RS method [11] and sample pair analysis [12] each exploit the statistical traces that replacement leaves in pixel values. Later work moved to feature-based classifiers [13] and convolutional networks [14]. On the embedding side, adaptive schemes concentrate changes where an image is already noisy or textured [15], and learned methods spread a secret image across all bits of a cover instead of confining it to the lowest ones [16]. Zielińska et al. give a broader overview of trends in the field [17].

1.3 Motivation and Contributions

This work was motivated by a practical need for small, readable LSB implementations in which the trade-off between capacity and visual fidelity can be observed directly. The contributions are as follows.

- An LSB text-in-image encoder and decoder built on OpenCV, using 8-bit character codes and an end-of-message delimiter.
- An image-in-image merge built on Pillow, with a comparison of two bit-allocation schemes (4/4 and 6/2) for dividing each 8-bit channel between cover and secret.
- An account of what the experiments show and do not show, including a storage-format confound in the image trials.

Section 2 describes the methods, Section 3 presents and analyses the results, and Section 4 concludes with future work.

2. METHODOLOGY AND IMPLEMENTATION

2.1 Environment

Both programs were written in Python and run in Google Colab, with images read from and written to Google Drive. The text experiment uses OpenCV [18] and NumPy [19]. The image experiment uses Pillow [20]. The source code is available at https://github.com/toranvathani/ProjectonSteganography.

2.2 Text-in-Image Embedding

The cover is a colour photograph loaded with cv2.imread, which returns a three-channel array. OpenCV stores channels in blue, green, red order, although the code names the variables r, g and b. This does not affect correctness, because the encoder and the decoder visit the channels in the same order. For a cover of height H and width W, using one bit per channel, the capacity is

$$C = (H \times W \times 3) / 8 \ \text{bytes} \quad (1)$$

For the cover used in the text experiment, the program reported a capacity of 94,357 bytes. The encoder proceeds as follows.

- Read the message and reject it if it is empty or longer than C.
- Append the delimiter *, then convert every character to an 8-bit string with format(ord(c), "08b") and join the strings into one bit sequence of length L = 8(n + 5) for an n-character message.
- Visit pixels in row-major order and, within each pixel, the three channels in array order. For each channel, drop the last bit of its 8-bit binary string, append the next message bit, and convert the result back to an integer.
- Stop modifying pixels when all L bits are written, then save the array with cv2.imwrite to a PNG file.

PNG is a lossless format, so the modified low-order bits survive storage. A lossy format would rewrite them. The test message was the 13-character string MynameisToran. With the delimiter, the bit sequence was 144 bits long and occupied 48 pixels at three bits per pixel. All other pixels were left unchanged.

The decoder walks the stego image in the same order and collects the last bit of every channel value. It groups the bits into bytes, converts each byte to a character, and stops as soon as the last five characters are *. The text before the delimiter is the message. The delimiter approach is simple but has two weaknesses: a message that itself contains * is cut short, and a decoder applied to an image with no payload keeps reading until the pixel array ends.

2.3 Image-in-Image Embedding

The second program hides a secret image inside a cover image. Both are opened with Pillow. The secret image (nature.png, 455 × 463 pixels) was resized to 300 × 200 pixels with resize((300, 200)), which does not preserve its aspect ratio. The cover (nature1.png) is 525 × 355 pixels. The merge function refuses to run if the secret is larger than the cover in either dimension.

Each pixel of both images is converted to three 8-bit binary strings. Where the cover has a pixel but the secret does not, the secret is treated as black, (0, 0, 0). If c and s are the 8-character channel strings of the cover and the secret, and k is the number of cover bits retained, the merged string is c[:k] + s[:8−k]. In integer form, with cover value c and secret value s,

$$m = (c \ \text{AND} \ (0xFF << (8-k))) \ \text{OR} \ (s >> k) \quad (2)$$

Extraction takes the low 8−k bits of each merged channel and pads them with k zeros on the right:

$$s' = (m \ \text{AND} \ (2^{(8-k)} - 1)) << k \quad (3)$$

The unmerge function also tracks the last non-black pixel in order to crop the output to the region that holds the payload.

First scheme, 4-bit/4-bit (k = 4). The initial implementation kept the top four bits of the cover and stored the top four bits of the secret in the bottom four positions. Recovery used the low four bits followed by 0000. The merged image was saved as merged1.jpg, and the unmerge function was run on that file after reading it back from disk.

Second scheme, 6-bit/2-bit (k = 6). After inspecting the first result, the split was changed to keep six cover bits and embed two secret bits. Recovery used the low two bits followed by 000000. In this trial the unmerge function was applied to the merged image object held in memory, not to a file read back from disk.

The worst-case per-channel error of each scheme, before any file compression, follows from the bit counts and is shown in Table -1.

Table -1: Bit allocation and worst-case per-channel error (values 0 to 255)

Scheme	Cover bits kept	Secret bits stored	Max cover error	Max secret error
4/4	4	4	15	15
6/2	6	2	3	63

Keeping more cover bits protects the cover and degrades the secret. The reverse also holds.

3. EXPERIMENTAL RESULTS AND ANALYSIS

3.1 Setup

Each experiment was run once on a single input. Evaluation was by visual inspection, and no numerical image-quality metric was computed. Table -2 lists the parameters.

Table -2: Experimental parameters

Item	Text experiment	Image experiment
Cover	Desert landscape photograph (download.jpg)	Sunset pier photograph (nature1.png), 525 × 355 px
Payload	MynameisToran plus * (144 bits)	Cracked-earth photograph (nature.png), 455 × 463 px, resized to 300 × 200 px
Capacity	94,357 bytes (reported by the program)	Not applicable
Library	OpenCV, NumPy	Pillow
Saved as	PNG	JPEG (merged1.jpg, merged2.jpg)

3.2 Text Embedding

The encoder accepted the message and reported success. The original and the stego photograph looked identical when placed side by side (Fig -1). This is the expected outcome: 144 of the several hundred thousand channel values in the cover were eligible to change, and each changed by at most one intensity level. Decoding the saved PNG returned the original string MynameisToran. The decoder must read the same file that the encoder wrote. Applied to any other image, it reads the low-order bits of ordinary pixel data and returns unreadable characters, because it has no way to tell whether a payload is present other than by finding the delimiter.



Fig -1: Original image (left) and image after encoding the text (right).

3.3 Image Merge with the 4-bit/4-bit Split

The merged image looked like the cover. The image recovered from merged1.jpg did not look like the secret. It was covered in coloured speckle noise (Fig -2). The noise extended across the whole frame, including the area outside the 300 × 200 region, which carried no secret bits and should have been recovered as black.

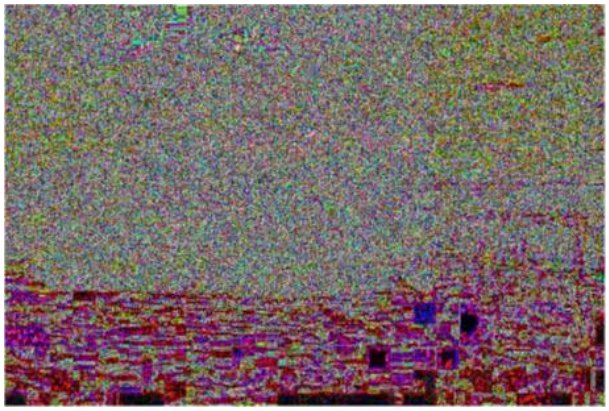

Fig -2: Image recovered from the 4-bit/4-bit merge, read from merged1.jpg.

The extra secret bits are not a satisfactory explanation. With four secret bits kept, the recovered value differs from the original by at most 15 levels per channel (Table -1), which would give a coarse but recognisable picture. Noise in a region that was black by construction points instead to a change in the low-order bits after embedding. The merged image was saved as a JPEG, and JPEG compression quantises transform coefficients and does not preserve individual low-order bits [21]. Because the unmerge step re-read that file, a payload stored in the low four bits would be corrupted. The 4/4 experiment was not repeated with a lossless format, so this is the most probable explanation and not a measured result.

3.4 Image Merge with the 6-bit/2-bit Split

With the 6/2 split, the merged image was visually indistinguishable from the cover (Fig -3c). The recovered image showed the cracked-earth scene, including the

horizon and the cloud layer, in the top-left 300 × 200 region where the secret had been placed. The rest of the frame was black, as expected (Fig -3d).

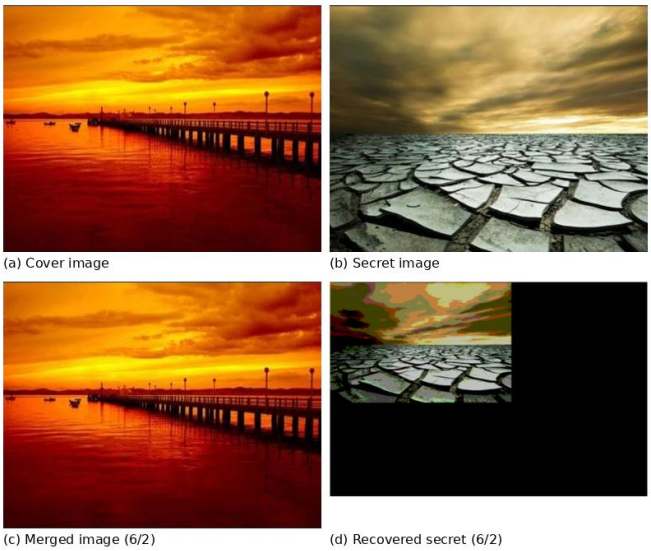

Fig -3: (a) cover, (b) secret, (c) 6/2 merged image, (d) secret recovered from the 6/2 merge.

The recovery is still coarse. Only two bits per channel survive, so each channel can take only the values 0, 64, 128 and 192 (Equation 3 with k = 6). Smooth gradients in the sky turn into flat bands of colour, and the colours shift. Because the discarded bits are filled with zeros, every recovered value is biased downward. A mid-range fill value would reduce this bias. It was not tested.

3.5 Analysis

Taken at face value, the 6/2 split gave a better recovered image than the 4/4 split. The reason needs care, because the two trials differ in more than the split. The 4/4 recovery came from a JPEG file on disk. The 6/2 recovery came from the in-memory image. The two secret bits of the 6/2 scheme occupy the lowest two bit positions and would be just as exposed to JPEG compression as the four bits of the 4/4 scheme. The in-memory path therefore probably accounts for much of the difference. A fair comparison would pass both schemes through the same storage pipeline.

The experiments do support three narrower points. LSB replacement leaves a cover visually unchanged for both a short text payload and an image payload in part of the frame. Reducing a secret image to two bits per channel costs visible tonal detail. And the storage format must be lossless if a payload held in low-order bits is to be recovered.

3.6 Limitations

One cover and one payload were tested per experiment. No peak signal-to-noise ratio, structural similarity or histogram comparison was computed, so the claims about imperceptibility rest on human viewing alone. Embedding is sequential and unkeyed, so anyone who knows the method can extract the payload. The text payload is stored in plaintext. The encoder compares the message length with the capacity before the five-character delimiter is added, so a message within five characters of the limit would overflow. Plain LSB replacement leaves statistical traces that detectors can find [10]–[12], and no detector was run against the output.

4. CONCLUSION AND FUTURE SCOPE

Two LSB steganography programs were implemented in Python: one hides an ASCII message in a photograph using OpenCV, and the other hides a resized image inside another using Pillow. The text encoder left the cover visually unchanged and the decoder recovered the 13-character message from the saved PNG. In the image experiment, a 6-bit/2-bit split gave a recognisable but coarsely banded recovery, while a 4-bit/4-bit split gave a noisy one. The noise in the 4/4 trial is probably due to JPEG storage and not to the bit split, although this was not tested directly.

The following work would make the study more rigorous and the tools more useful.

- Run both splits, and others such as 5/3 and 7/1, through the same lossless pipeline on a set of images, and report PSNR [22] and structural similarity [23] for the cover and the recovered secret.
- Apply the chi-square, RS and sample pair detectors [10]–[12] to the stego images and report detection rates.
- Choose embedding positions with a pseudo-random sequence seeded by a shared key.
- Encrypt the message before embedding and store its length in a header instead of relying on the * delimiter.
- Preserve the aspect ratio when resizing, use a mid-range fill for discarded bits, and test the effect of recompression and rescaling.
- Compare against adaptive embedding [15] and learned hiding [16], and extend the approach to audio and video covers.

ACKNOWLEDGEMENT

The author thanks Vellore Institute of Technology for the academic environment in which this work was carried out as a project for the course Cyber Security (BCSE410L).

REFERENCES

1) N. Provos and P. Honeyman, "Hide and seek: An introduction to steganography," IEEE Security & Privacy, vol. 1, no. 3, pp. 32–44, May–Jun. 2003.

2) N. F. Johnson and S. Jajodia, "Exploring steganography: Seeing the unseen," Computer, vol. 31, no. 2, pp. 26–34, Feb. 1998.

3) F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn, "Information hiding A survey," Proceedings of the IEEE, vol. 87, no. 7, pp. 1062–1078, Jul. 1999.

4) R. J. Anderson and F. A. P. Petitcolas, "On the limits of steganography," IEEE Journal on Selected Areas in Communications, vol. 16, no. 4, pp. 474–481, May 1998.

5) J. Fridrich, Steganography in Digital Media: Principles, Algorithms, and Applications. Cambridge, U.K.: Cambridge University Press, 2009.

6) I. J. Cox, J. Kilian, F. T. Leighton, and T. Shamoon, "Secure spread spectrum watermarking for multimedia," IEEE Transactions on Image Processing, vol. 6, no. 12, pp. 1673–1687, Dec. 1997.

7) L. Caviglione, M. Cabaj, and W. Mazurczyk, et al., "The new threats of information hiding: The road ahead," IT Professional, vol. 20, no. 3, pp. 31–39, May–Jun. 2018.

8) C.-K. Chan and L. M. Cheng, "Hiding data in images by simple LSB substitution," Pattern Recognition, vol. 37, no. 3, pp. 469–474, Mar. 2004.

9) A. Cheddad, J. Condell, K. Curran, and P. Mc Kevitt, "Digital image steganography: Survey and analysis of current methods," Signal Processing, vol. 90, no. 3, pp. 727–752, Mar. 2010.

10) A. Westfeld and A. Pfitzmann, "Attacks on steganographic systems," in Information Hiding (IH 1999), Lecture Notes in Computer Science, vol. 1768. Berlin, Germany: Springer, 2000, pp. 61–76.

11) J. Fridrich, M. Goljan, and R. Du, "Detecting LSB steganography in color and gray-scale images," IEEE MultiMedia, vol. 8, no. 4, pp. 22–28, Oct.–Dec. 2001.

12) S. Dumitrescu, X. Wu, and Z. Wang, "Detection of LSB steganography via sample pair analysis," IEEE Transactions on Signal Processing, vol. 51, no. 7, pp. 1995–2007, Jul. 2003.

13) J. Fridrich and J. Kodovský, "Rich models for steganalysis of digital images," IEEE Transactions on Information Forensics and Security, vol. 7, no. 3, pp. 868–882, Jun. 2012.

14) J. Ye, J. Ni, and Y. Yi, "Deep learning hierarchical representations for image steganalysis," IEEE Transactions on Information Forensics and Security, vol. 12, no. 11, pp. 2545–2557, Nov. 2017.

15) V. Holub, J. Fridrich, and T. Denemark, "Universal distortion function for steganography in an arbitrary domain," EURASIP Journal on Information Security, vol. 2014, art. no. 1, 2014.

16) S. Baluja, "Hiding images in plain sight: Deep steganography," in Advances in Neural Information Processing Systems 30 (NIPS 2017), Long Beach, CA, USA, 2017, pp. 2066–2076.

17) A. Zielińska, W. Mazurczyk, and K. Szczypiorski, "Trends in steganography," Communications of the ACM, vol. 57, no. 3, pp. 86–95, Mar. 2014.

18) G. Bradski, "The OpenCV library," Dr. Dobb's Journal of Software Tools, 2000.

19) C. R. Harris et al., "Array programming with NumPy," Nature, vol. 585, pp. 357–362, Sep. 2020.

20) A. Clark et al., "Pillow (PIL Fork) documentation," [Online]. Available: https://pillow.readthedocs.io

21) G. K. Wallace, "The JPEG still picture compression standard," IEEE Transactions on Consumer Electronics, vol. 38, no. 1, pp. xviii–xxxiv, Feb. 1992.

22) Q. Huynh-Thu and M. Ghanbari, "Scope of validity of PSNR in image/video quality assessment," Electronics Letters, vol. 44, no. 13, pp. 800–801, Jun. 2008.

23) Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600–612, Apr. 2004.

BIOGRAPHY

Toran V Athani (Reg. No. 21BCE2405) is a graduate of the B.Tech program in Computer Science and Engineering at Vellore Institute of Technology, Vellore, India. This paper is based on his project for the course Cyber Security (BCSE410L) completed during the program.