

**Data hiding and extraction using pseudo-random generation and cover image replication****Mohammad K. Al-Laham<sup>a</sup>, Nameer N. El-Emam<sup>b</sup> and Kefaya Qaddoum<sup>c\*</sup>**<sup>a</sup>*Computer Science Faculty of Information Technology, Philadelphia University, Jordan*<sup>b</sup>*Computer Science Faculty of Computer Science and Informatics, Amman Arab University, Jordan*<sup>c</sup>*Computer Science Faculty, Concordia University, Canada***ABSTRACT***Article history:*

Received June 26, 2024

Received in revised format August 28, 2024

Accepted October 12 2024

Available online

October 13 2024

*Keywords:**Steganography**Multi cover image**Least Significant Bit**Pseudo-Random Generation**Load balance*

This paper introduces a new algorithm in Steganography for concealing secret messages or images within digital images. The algorithm is designed to produce stego images that can be transmitted to recipients without detection by potential attackers, thereby ensuring secure communication channels. The proposed algorithm employs a multi-level randomization technique to embed data within randomly selected cover images, with each byte of the secret image distributed across multiple cover images. This approach contrasts with conventional methods that hide data within a single cover image. Moreover, the algorithm incorporates a load-balancing priority system, a critical feature that ensures uniform stego image quality across the dataset. This strategic approach minimizes variations in Peak-Signal-to-Noise-Ratio (PSNR) values, contributing to consistent performance during data hiding and extraction processes and enhancing communication security. The security and recoverability of the secret image are further improved by a simplified Cipher key system based on SHA-256, which facilitates pseudo-random number generation. This system ensures that the hidden image can be recovered at the receiver's end, even in the face of potential attacks. Experimental results demonstrate comparable PSNR quality to existing methods, particularly when utilizing equal total resolution to deep hiding algorithms. Notably, the proposed algorithm offers an alternative to encryption by leveraging randomization, thereby complicating data extraction for potential attackers by distributing data across multiple images with a randomly generated cipher key.

© 2025 by the authors; licensee Growing Science, Canada.

**1. Introduction**

Steganography conceals secret information within ordinary objects or messages, contrasting with cryptography, which focuses on data encryption (Ansari et al., 2020). Steganographic algorithms typically involve encryption or encapsulation for added security. Security and statistical analysis pose significant challenges in steganography, causing multiple encryption and compression phases to secure data before concealment (Płachta et al., 2022). LSB-based algorithms in single-cover images are vulnerable to statistical attacks. The Least Significant Bit LSB, commonly used for its simplicity, embeds hidden messages within image pixels' least significant bit without perceptibly altering the image (Al-Shaaby & AlKharobi, 2017). This study endeavors to pioneer advancements in steganalysis by addressing critical vulnerabilities in existing data-hiding algorithms. Specifically, the objectives are to fortify defenses against adjacent pixel calculations, known attacker exploits, and payload-based statistical attacks, elevating the levels of security inherent in the data-hiding process. Notably, this work proposes replacing conventional encryption layers with a randomized obfuscation step during data hiding, simplifying algorithmic complexity while bolstering data security. This research presents a novel data-hiding algorithm that vertically conceals data, utilizing multiple cover images simultaneously with randomization to disrupt predictable patterns. Achieving comparable PSNR quality with approximately 75% of the resolution of a single cover image and superior PSNR quality with multiple

\* Corresponding author

E-mail address [N.emam@aau.edu.jo](mailto:N.emam@aau.edu.jo) (K. Qaddoum)

ISSN 2561-8156 (Online) - ISSN 2561-8148 (Print)

© 2025 by the authors; licensee Growing Science, Canada.

doi: 10.5267/j.ijdns.2024.10.005

cover images of identical total resolution demonstrates high-quality data hiding feasibility. Each byte of the secret image is distributed across various independent stego images, preventing partial reconstruction without access to all stego images.

The paper is structured into sections covering a literature review, theoretical foundations, algorithm architecture, empirical results, and conclusions with recommendations. The subsequent sections of this paper are structured as follows: Section two delves into relevant literature, while section three elucidates the theoretical underpinnings of Steganography and associated testing standards. Section four comprehensively details the architecture and operational intricacies of the proposed algorithm, augmented by pseudo-code illustrations. Section 4 presents empirical results and algorithmic analyses, culminating in Chapter 5's conclusion and actionable recommendations.

## 2. Related works

In this section, we will review various research works in Steganography and compare them with the proposed data hiding and extraction algorithms.

### 2.1 Data hiding

Al-Shaaby and AlKharobi (2017) introduced a new method of hiding secret information in images, audio, or video by combining cryptography and steganography. In this approach, the message is encrypted using the AES algorithm, and the key is hashed using SHA-2 to prevent attacks. The approach is fundamentally strong but could be improved by detailing key management, specifying the steganographic technique, and evaluating performance and robustness. Elshare and El-Emam (2020) strengthened the Deep Hiding Extraction Algorithm (DHEA) to improve the quality and security of stego images by incorporating advanced encryption techniques and adaptive data embedding strategies. The approach could be improved by specifying the encryption techniques and adaptive strategies used, quantifying image quality improvements, conducting a detailed security evaluation, and including performance metrics. Alabaichi et al. (2020) explored the utilization of 3D Chebyshev polynomial mapping to enhance the efficiency and security of data-hiding algorithms, achieving robustness against various steganalysis techniques. The approach here mapped and enhanced the hiding algorithms but still lacked comprehensive security and robustness evaluations. Elharrouss et al. (2020) proposed a novel data-hiding scheme based on K-LSB embedding and replication techniques, aiming to maximize the data-hiding capacity while minimizing the perceptual impact on stego images. The paper listed an improvement of the efficiency and security of data hiding while it lacks a thorough robustness evaluation. Haider et al. (2024) introduced a novel pseudo-random number generator (PRNG) technique for discrete data hiding, providing enhanced randomness and unpredictability in the embedding process. The study does not include performance metrics such as computational efficiency, which are crucial for real-world applications where processing time and resources are limited. Nie et al. (2019) employed the knight tour algorithm to facilitate pixel selection in data-hiding processes, enhancing the randomness and unpredictability of data embedding. Zhu et al. (2022) used Residual Convolutional Neural Networks to increase the number of layers used for data hiding and extraction. Liu et al. (2022) utilized an encoder to analyze the reference image, aiming to improve the quality of the resulting generated images and generate the cover image based on the structure and texture of a reference image. Their mapping technique could be improved by providing detailed implementation and performance metrics. Abuali et al. (2024) presented an innovative approach to multi-level security enhancement using steganography. Their method combines Dynamic Least Significant Bit (DLSB) steganography with Wavelet Obtained Weights (WOW) steganographic algorithms, creating a sophisticated system for concealing information within non-obvious data. The paper did not cover exploring the integration of advanced encryption algorithms, and evaluating the approach's performance across different types of multimedia.

### 2.2 Extraction Algorithms

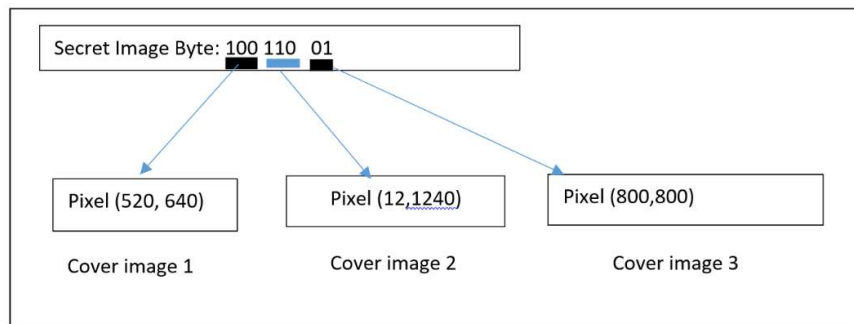
Wei et al. (2022) trained a generator to create synthetic images from authentic images and incorporated several discriminators to generate statistically indistinguishable step images. Ahmad et al. (2021) presented an improved variant of the DHEA algorithm, focusing on enhancing stego image quality and security through advanced encryption mechanisms and dynamic data embedding strategies. Pei et al. (2020) optimized the capacity and quality of stego images through payload-based calculations, resulting in improved payload capacity and visual quality of stego images. The authors could explore optimizing the capacity and quality of images. In the field of steganalysis attack detection, various methodologies were reviewed. Wang et al. (2020) presented an algorithm to predict the locations of the payload in a stego image to retrieve the original data. Yang et al. (2019) conducted steganalysis attacks to evaluate the weaknesses in utilizing multiple stego images alongside LSB to hide an SM from attackers. Sun et al. (2019) developed advanced algorithms employing statistical analysis and machine learning techniques to detect subtle alterations caused by data-hiding processes, thereby enhancing the effectiveness of steganalysis attack detection.

## 3. Data Hiding using PRNG and Load Balancing

Random Number Generation will generate a sequence of numbers that other parts of the algorithm will use to alter the data-hiding algorithm flow. To ensure the random number generator does not cause issues with data recovery, we will construct

the random number generator in a simplified way inspired by the SHA-256 encryption algorithm. A cipher key will be used to generate the numbers the random number generator uses. This way, a pseudo-random number generator (PRNG) is created, as it will always produce the same sequence of output when the same Cipher Key is provided. An incorrect Cipher Key will produce a completely different sequence of numbers. Randomization with a Cipher Key can hide sequence data, making it difficult to guess the start sequence of the least significant bits. Hiding data across all three-pixel channels can also allow more bits to be hidden per pixel than a technique that only uses one channel. However, using only one channel per pixel can reduce the impact on stego images, leading to lower differences from a cover image.

Unlike some other methods, such as Ahmad et al. (2021), Elshare and El-Emam (2020), and Baluja (2017) are based on deep data hiding, the new algorithm randomizes the application of LSB across multiple images to split the data stream in an untraceable manner. This approach also allows various cover images, increasing data storage capacity and reducing the need to compress and hide multiple bits within a single image. However, the new algorithm does not currently implement a similar phase, as the focus was on enhancing any LSB-based algorithm with access to multiple images and increased security using a Cipher Key. We will use various cover images instead of one cover image to improve the random number generators' further' and Cipher Keys' impact on the data hiding. By distributing the data randomly across multiple images where each bit of a pixel from the original image could end up in an entirely different cover image, we introduce a new layer of complexity to the recovery of the image. The order of the hidden bits will be scrambled across multiple images. A pseudo-random Number Generator (PRNG) will determine the positions to hide data in the images and when to swap the current cover images during data hiding. Therefore, changing the current image randomly at random periods will make it impossible to brute force a sequence of images to recover the data even if the attacker knows the presence of Steganography. Another security feature introduced by randomization using the Cipher Key over an LSB technique is that the data will follow no pattern that can be brute-forced to recover it. This is because, without the cipher key, the attacker will not build up the random number generator sequence to retrieve the correct bits in the same order necessary to replicate the bytes of the original image. For example, the following figure demonstrates data hiding randomly across multiple photos.



**Fig. 1.** Byte distribution into multiple cover images

In Fig. 1, the first three parts of the current BYTE will be hidden in cover image 1, the next 3 bits will be hidden in cover image number 2 instead, and the last 2 bits will be hidden in cover image number 3. When the random number generator is added at the top, it will not follow the same order of cover images. For example, with a different Cipher key, the sequence of bits might change to 2 bits in cover image 2, followed by 3 pixels in Cover image 1, and finally, the last 3 bits in cover image 3. Assuming the same order of images is used from the previous example to recover the data, the byte will be 0111001 instead of 10011001. Due to the random number generation, the sequence of the cover images that hide data will not be static. Each time data is hidden, we will hide the next 3 bits into the three channels of the next byte from a random cover image. This is more secure than hiding data across multiple images, but each image is independent of the other, hiding the data into independent segments of different images. By design, the new algorithm will hide data across multiple cover images. However, the data follow a strict order of those cover images, which forms dependency between them and makes it impossible to recover segments of the original image without following the correct sequence data hidden.

One problem introduced by randomization must balance the secret image into the cover images. We introduce a weight/priority component to the cover images. The more times the random number generator selects a cover image, the less likely it will be chosen again. This gives other images a higher priority when selecting the randomizer next. Without this load balancing, the process hides too many bits in some photos while hiding little to no data in others. This is important as the more modified an image, the more vulnerable it will be to steganalysis. With load balancing, the algorithm minimizes the changes to each cover image. It adds an extra layer of randomization, as the algorithm will seemingly stop following a pattern at random intervals. It has an obfuscated hiding pattern because of missing information without a cipher key to determine when to drop the current random number due to the weight system. To prevent an infinite loop due to the priority system, the algorithm will have to deactivate cover images once they cannot be used to hide more data as the payload capacity limit is reached. When an image is deactivated, the algorithm will not skip cover images as often by providing an increased chance to pass priority tests. The more cover images are deactivated, the less likely the remaining cover images are to fail the priority

load balancing checks. The algorithm supports randomizing the distance between selected pixels to avoid sequential LSB hiding in the cover images; this will reduce the max capacity of each image but makes it impossible to retrieve all data by brute-forcing the first pixel in a sequence. Replication means rebuilding the original secret image using the data within the cover images. Because of randomization and encryption, we use the random number generator to gather information on rebuilding the original image instead of simply hiding data in a single image. Note that we avoided working on the alpha channel as it is less commonly used than the standard color channels. It would be straightforward for steganalysis tools to know if an image was modified if the alpha channel is randomly changing even by 1 bit to color bytes. Colors in photos are subject to shadows and gradients, which might explain the changes in the bits, which is not the same for the alpha channel, as the alpha in most images is 0 or 255. It will be possible to hide more data in images with an alpha channel, but the opaque parts will show something wrong with the channel in the image.

#### 4. Hiding and Extraction Algorithms

##### 4.1 Data Hiding Algorithm

The following sections detail the definitions and classes used in this algorithm. A Custom image class is used to increase control on cover images and keep track of important information between different functions and functions.

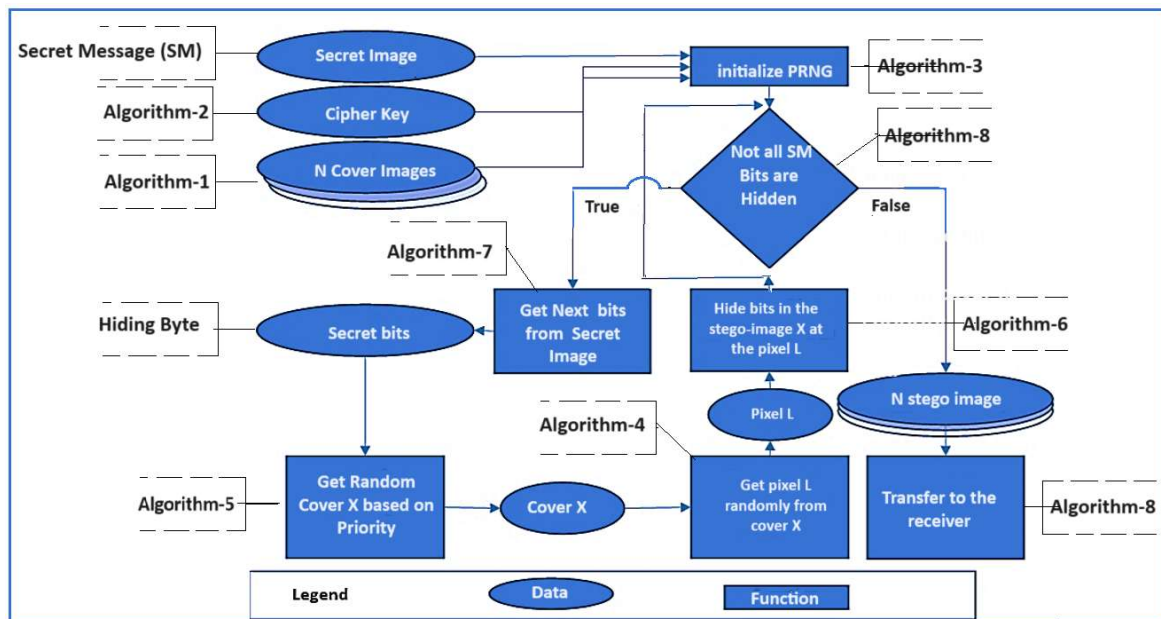


Fig. 2. Data Hiding Architecture

Fig. 2 demonstrates the algorithm's data hiding and replication stages. The test application uses standard LSB, but more complex steganography algorithms and noise reductions can be implemented to improve data hiding further. The goal is to implement randomization at multiple cover images while hiding the input randomly across numerous images. There is no way to independently retrieve the secret image hidden in each image. Algorithm 1 sets the information needed to start data hiding in each cover image. The Cipher key (CK) generates the initial randomization matrix. This cipher key is only used to generate the initial values; the randomizer will loop to the start of the generated matrix when it reaches the End, and a bigger CK will create more random numbers to reduce looping in the function. A smaller CK requires less memory and runs faster but might lead to a visible pattern without randomization applied at the pixel selection level. This way of handling enables PRNG, where the same CK will consistently reproduce the same random input for the application to do the same actions in the same way between data hiding and data extraction algorithms.

**Algorithm 1. CustomImage****Function CustomImage(Inputimage image)**

```

{
/* Let  $I_n$  be the  $n^{\text{th}}$  CI
Let  $I^{\text{initial}x}$  and  $I^{\text{initial}y}$  be the coordination of the first pixel where data hiding starts on a cover image.
Let  $I^{\text{hiding}x}$  and  $I^{\text{hiding}y}$  be the coordination of the next pixel that will be used for data hiding.
Let  $I^{\text{prio}}$  be the current priority of a CI that can be selected or reselected for more data hiding when its priority is high.
Let  $i^{\text{dir}}$  determines the data hiding or extraction direction (horizontally or vertically).
Let  $I^{\text{inactive}}$  be a testing variable to check CI it is still enabled for data hiding.
Let  $\text{InputImage}^{\text{width}}$  be the width of input image
Let  $\text{InputImage}^{\text{height}}$  be height of input image
*/

 $I^{\text{initial}x} \leftarrow \text{InputImage}^{\text{width}} / (\text{PRNG\_NextNumber} + 1)$  // Random start position X
 $I^{\text{initial}y} \leftarrow \text{InputImage}^{\text{height}} / (\text{PRNG\_NextNumber} + 1)$  // Random start position Y
 $I^{\text{hiding}x} \leftarrow (I^{\text{initial}x} + 1) \% \text{InputImage}^{\text{width}}$ 
 $I^{\text{hiding}y} \leftarrow (I^{\text{initial}y} + 1) \% \text{InputImage}^{\text{height}}$ 
 $I^{\text{prio}} \leftarrow 4$  // Priority is saved on a CI custom image object level
 $I^{\text{active}} \leftarrow \text{true}$  // CI start actively to allow storing data in them
 $I^{\text{dir}} \leftarrow ((\text{PRNG\_NextNumber}() + 1) \% 2 == 0)$  // Assign a random direction to the Image
} // end of CustomImage

```

When CK is not provided, a default value that was hardcoded in the algorithm can be used instead to avoid the need for communication about the CK and keep it implicit, with no way to find it besides having the correct algorithm to extract the CK. This is similar to encryption but without modifying the content of the cover images; this distinction is for minimizing the changes on any single image when unnecessary. Furthermore, because of the randomization in the algorithm, it is not necessary to encrypt the data as functionally; the two approaches achieve the same goal: blocking attackers from retrieving any data without the correct sequence to retrieve the bits.

**Definition 1: CipherList:**

The list of integers that are generated from the CK is defined in the CipherList Eq(1),

$$\text{CipherList} = \{i \text{ is the } i^{\text{th}} \text{ digit in the CipherList}\} \quad (1)$$

When a list of integers is generated from the CipherKey, the next digit of the current integer determines the randomization outcome whenever a random number generator-dependent action is triggered. This forms a random number matrix as the outer list contains integers, and the process treats the integers themselves as an array.

**Definition 2:** Let  $i_{\text{current\_cover}}$  be a variable that tracks the current cover image index used for data hiding.

**Definition 3:** Let  $i_{\text{inactive\_covers}}$  be the number of deactivated cover images used to keep track of the current number of deactivated cover images; an image is deactivated when it has no more space for data hiding if all cover images are inactive. The user can load more cover images to increase total capacity or stop execution and start over with more cover images.

**Definition 4:** Let  $i_{\text{bonus\_priority}}$  be the bonus priority added to cover images when deactivated. Bonus priority is added to make it easier for cover images to pass the PRNG checks when selecting the following cover image for data hiding.

**Definition 5: Image progress variables:**

The pixel selection optional variables can be used to add randomization during pixel selection in the cover images. Let  $I^{\text{stepsize}}$  be the step size used in addition to the random number generator to determine the space to the next pixel in the cover image used for data hiding. Let  $I^{\text{stepmultiplier}}$  be a multiplier that is applied with randomization to generate higher distances between pixels used for data hiding; higher multiplier values increase the distance between pixels in the cover image but limit the max number of usable pixels for data hiding, reducing the max capacity of each image. Bigger cover images allow larger step sizes and multipliers than small images. Let  $I^{\text{main\_step\_size}}$  and  $I^{\text{main\_step\_multiplier}}$  do the same, but in the main hiding direction, the direction  $I^{\text{dir}}$  is determined Algorithm 1.

**Definition 6:** Let Cover Images be a set of CI of the size  $x$  that will be used to hide the secret image, see Eq. (2):

$$\text{CoverImages}^x = \{i \in (0, \infty)\}, \quad (2)$$

where the index ( $i$ ) of the cover images starts from 0 to infinity, the index  $i=0$  indicates the first cover image loaded by the process. Each loaded cover image will result in a stego image SI when modified directly from this set.

**Definition 7:** Initilaize\_Cipher

Let initialize\_Cipher be the function that generates the cipher list from the CK, see Eq. (3):

$$\text{Initialize\_Cipher: CipherKey} \rightarrow \text{CipherList} \quad (3)$$

Algorithm 2 is applied to initialize Cipher, which is the function that generates the CipherList used by other functions in the algorithm. Depending on the length of the CipherKey, more complex number groups will be generated and stored in the CipherList.

**Algorithm 2. Initilaize Cipher**

```

Function initialize_Cipher(String CK) // CipherList initialization used for PRNG
{
// Assign groups of 12 digits in each index of the cipherlist, larger numbers higher memory
Let integer Group_size ← Minimum(CKLength/3, 12);
Let String CipherCopy ← Concat(CipherKey, CipherKey); // Increase size of CK
for each character index ck_pos | ck_pos ∈ (0, CipherKeyLength )
    CipherListLength+1 ← Square Value ( Substring(CipherCopy(ck_pos, group_size));
}

```

**Definition 8:** PRNGNextNumber: The random number generator function forms the pseudo-random number generator (PRNG), whose output is used by the rest of the algorithm. It returns the following number stored in the cipher list generated during the CK initialization function. The function starts from the first randomly generated number when no numbers are in the cipher list.

Let PRNG\_NextNumber be the function that returns the following random number stored in CipherList. Variables  $i\_current\_cipher$  and  $current\_cipher\_char$  are the index of the subsequent number the function will return from the CipherList; see Eq. (4).

$$\text{PRNG\_NextNumber: } i\_current\_cipher \times current\_cipher\_char \rightarrow \text{CipherList} \quad (4)$$

Algorithm 3 returns the next digit in the current integer group in CipherList. This function is called to produce a randomized outcome. When no more digits exist in the CipherList, the function starts over from the first generated digit.

**Algorithm 3. PRNG NextNumber**

```

Function PRNG_NextNumber() returns Integer {
    /* Pseudo Random Number Generator CipherList, i_current_cipher / and current_cipher_char are global algorithm variables */
    Let integer current_digit ← CipherListi_current_ciphercurrent_ciph_char // Store current digit
    Let integer current_cipher_char ← current_cipher_char + 1 // increment index
    if (current_cipher_char >= CipherListi_current_cipherCount) {
        /* reset index when out of group length progress to next group of digits in CipherList */
        i_current_cipher ← (i_current_cipher + 1) mod CipherListCount
        current_cipher_char ← 0; // Reset digit index in each group to starting position
    }
    Return current_digit; // current digit contains the next random number from cipherList
}

```

**Definition 9:** The Next\_Pixel Function is applied when working on cover and stego images. Two functions handle updates on the cover image when producing a stego image; see Eq. (5).

$$\text{NextPixel: } CI_{ij}^p \times \text{PRNG\_Next\_Number} \rightarrow CI_{i',j'}^{p'} \quad (5)$$

where  $i$  and  $j$  are the image's HidingX and HidingY data hiding positions, respectively,  $p$  is the cover image's priority (prior).

This NextPixel function is implemented at the custom image class level. Therefore, all objects of type customImage will have access to it. This way, it is possible to use the internal values of the custom image variable  $I^{prio}$  during execution without impacting other images or limiting the number of CI the algorithm can use. In the next step, size is used to randomize the distance between pixels during data hiding to stop the process from using a specific length between data in the cover image, adding complexity to the data recovery but reducing the concentration of data hidden in a single area within the cover image.

Algorithm 4 is applied to get the Next\_Pixel to determine the next pixel when hiding data in the current cover image. At the start, a priority test is done by comparing the result of the function PRNG\_Next\_Number with the cover image's priority, which must be passed, or the cover image cannot be used, and another cover image must be selected randomly. If the cover image is used, the priority is changed to make it less likely to reuse the same cover image in the following data hiding; this reduces the chances of sequentially hiding data over and over in the same image and improves data distribution across multiple images.

A cover image can be reused multiple times if it passes the priority test several times. The priority of the cover image will be reduced every time it is used in data hiding and increased whenever it fails the random number generator test.

#### Algorithm 4. Next\_Pixel

```

Function Next_Pixel () returns bool {
  /* Random Pixel Selection based on priority and direction I is the customImage calling this function, i_bonus_priority is bonus priority applied to all
  images when a cover image is deactivated */
  If (PRNG_Next_Number() > Iprior - i_bonus_priority) {
    i_bonus_priority ← i_bonus_priority + 1; // Increase priority when the image fails the PRNG priority test
    return false } // End of priority failure condition
  else {
    prior ← Iprior - 1 } // Decrease priority when the image passes the PRNG priority test
  /* Data hiding direction for the current cover image based on b_direction value in the object of the class custom image */
  if (Idir = true) {
    IHidingY ← IHidingY + next_step_size(); // Calculate distance to next pixel
    if (IHidingY >= IHeight) {
      IHidingY ← 0;
      IHidingX ← (IHidingX + next_step_main_size()) mod IWidth
    }
  }
  /*An image is deactivated when hiding data in it risks overwriting other secret image data; this way, the algorithm achieves lossless data hiding*/

  if (IHidingX == IinitialX AND IHidingY >= IinitialY) {
    Iactive ← false
    i_inactive_covers ← i_inactive_covers + 1
    i_bonus_priority ← i_bonus_priority + 1
  }
} // End of Idir = true
Else { // Idir = false
  IHidingX ← IHidingX + next_step_size()
  if (IHidingX >= IWidth) {
    IHidingX ← 0
    IHidingY ← (IHidingY + next_step_main_size()) mod IHeight
  }
  //End of else condition
  if (IHidingY == IinitialY AND IHidingX >= IinitialX) { // Data integrity condition to prevent overwriting existing SM data
    Iactive ← false // Deactivate the image to prevent overwriting of secret image data already in the stego-image
    Iinactive_covers ← i_inactive_covers + 1;
    i_bonus_priority ← i_bonus_priority + 1
  } // End of data integrity condition
} // End of Idir = false conditions
} // End of Next_Pixel

```

#### Definition 10:

Let NextStepSize and next\_main\_step\_size be the functions that calculate the distance between pixels in the cover image; see Eqs. (6-7).

$$\text{NextStepSize: PRNG\_NextNumber} \rightarrow (\text{PRNG\_NextNumber}) \bmod \text{stepsize} \times \text{stepMultiplier} \quad (6)$$

$$\text{next}_{\text{main\_step\_size}} : \text{PRNG\_NextNumber} \rightarrow (\text{PRNG\_NextNumber}) \bmod \text{main\_step\_size} \times \text{main\_step\_multiplier} \quad (7)$$

Let the image function NextStepSize be a function that returns a random digit to determine the minimum number of pixels between the last data-hiding pixel and the next position. For example, if NextStepSize is two, the process must skip at least the next two pixels plus a random number in the cover image before hiding data again. Stepsize determines the maximum distance between the current and the next pixel for data hiding. stepMultiplier increases the step size by allowing higher values than the range [0.9] in the function results. The process will hide data sequentially when stepsize and multiplier are one. Otherwise, gaps will be present between pixels because of the minimum distance limitation imposed by the function. Main\_step\_size and main\_step\_multiplier work similarly but use different amounts to support more complex data-hiding patterns than using the same random distance between pixels in both X and Y directions.

**Definition 11:** PRNG\_NextCover

Let PRNG\_NextCover be a function that performs load balancing and a PRNG check when the algorithm is performing cover image-level randomization; see Eq. (8).

$$\text{PRNGNextCover: } i_{\text{current\_cover}} \rightarrow \text{PRNGTest}, \quad \text{where PRNGTest} \in \{\text{true}, \text{false}\} \quad (8)$$

Before selecting a cover image, the nextPixel function of the current cover image object is called to perform a test comparing the following random number generated by Definition 10 and the priority of the current cover image. If the cover image passes the PRNGTest, the image will be used for data hiding; the function RNGnextCover returns true in this case. On the other hand, when a cover image fails the PRNGTest, the algorithm starts looking for the next suitable cover image for data hiding. Whenever a cover image fails the PRNGTest, the function will change the current cover image to the next active one in the cover images list. Once an active cover image is found, the function will repeat the random number comparison with the current cover image priority to determine if the image can be used for data hiding. The function recursively loops until an active cover image passes the random number generator test.

**Algorithm 5. PRNG\_NextCover**

```

Function PRNG_NextCover (i_current_cover) returns Boolean {
  /* PRNG Test on images i_current_cover is available to all functions in the algorithm */
  Let continue_Search ← true
  if (CoverImagesi_current_covernextPixel() == false) {
    /* Use custom image next_pixel to test PRNG priority Recursively search for the next active cover image when the current CI fails the PRNG
    priority check */
    if (inactive_covers = CoverImagesCount)
      load more cover images into CoverImages n
    while (continue_Search) { // Search for next active CI
      i_current_cover ← i_current_cover + 1 mod StegoImagesCount
      if CoverImagesi_current_coveractive = true then continue_search ← false
    }
    return PRNG_nextcover(i_current_cover) // Recursively call function
  }
  Return true // stops recursive calls when a cover image and pixel are found
}

```

Algorithm 5. PRNG\_NextCover is a recursive function that finds the next valid cover image for data hiding. A cover image is invalid if it is inactive or the next value from the random number generator fails the priority comparison. The call to the function NextPixel contains the implementations for random number generation and priority at a pixel level during data hiding. When all cover images are deactivated, the user can stop the algorithm or load more cover images to continue the data hiding from that step; newly loaded cover images will be initialized using the custom image constructor.

**Definition 12**

Let fillnextcover be a function to hide 3 bits from SM into {R, G, B} channels at  $i^{\text{th}}$  cover image selected randomly. This function should modify the  $i^{\text{th}}$  stego image for each hiding process at each channel, see Eq. (9).

$$\text{fillnextcover: } CI_i^R \times CI_i^G \times CI_i^B \times (SM_j)_{j=1,\dots,3} \rightarrow SI_i \quad (9)$$

In Algorithm 6, there is a call to the function in the definition of Algorithm 5. PRNG\_NextCover is used to swap to the next active cover image by performing random number and priority tests on the cover images until the following cover image passes the random number checks. After a cover image is found, the process will update the current image with the 3 bits forwarded to the function. The pseudo-code displayed hides the 3 bits from the secret image into three random pixels in 3 random cover images. PRNG\_NextCover() can be called three times instead of sending each color bit to hide into one specific or random channel.

The SetPixel is the function that modifies the current pixel of CI at the coordinates (X and Y) of the pixel at the channel (Color) of the  $i^{\text{th}}$  cover image  $CI_i$  to produce  $i^{\text{th}}$  SI, see Eq. (10).

$$\text{SetPixel: } X \times Y \times \text{Color} \times CI_i \rightarrow SI_i \quad (10)$$

where



**Algorithm 6. Swap next cover**

```

Function fillnextcover(CI,bit BitR,bit BitG,bit BitB) {
  Let HidingColor be of type Color // CI pixel where we need to perform LSB
  if (PRNG_NextCover() == true) // Call PRNG_NextCover to perform PRNG test and select next cover image and pixels
  {
    HidingColor ← CoverImagesicurrent_coveriHidingX,iHidingY // Retrieve Pixel to embed data in the CI
    Apply LSB on HidingColorR,G,B such that BitR,BitG and BitB are embedded in a random order
    SIicurrent_cover ← SetPixel(CIX,CIY,HidingColor,CIicurrent_cover)
  }
} // End of condition and function

```

**Definition 13: DataHidingWrapper**

Let DataHidingWrapper(.) be a function that sends the bits of the SM to the function that selects a random cover image and performs the actual data-hiding functionality. The data-hiding wrapper in Eq. (11) works with the Red (R), Green (G), and Blue (B) bytes of a pixel saved in the Color data type. HB is the three-channel bytes of the selected pixel from SM for hiding in CI. DataHidingWrapper hides all bits sent to it in random cover images by calling the function fill\_next\_cover, which will find the next cover image to use in the data-hiding process. For example, the three least significant bits from the three-color channels (RGB) will be sent to hide the 3 LSB bits in the subsequent cover images. It is possible to extend this function to do the cover image swap instead of independently for each color channel to store them in different cover images.

$$\text{DataHidingWrapper: SM}_{\text{HB}^{\text{R,G,B}}} \times \text{CI}^{\text{R,G,B}} \rightarrow \text{SI}^{\text{R,G,B}} \quad (11)$$

**Algorithm 7. DataHidingWrapper**

```

// DataHiding wrapper used to call the function that randomly picks a CI for data hiding
Function DataHidingWrapper(SMHBR,G,B, CI) // Color={R,G,B}
{
  Let HidingByteR,G,B ← SMHBR,G,B
  For each bit in HidingByte // 24 bits in each pixel represented by hidingbyte
    SI ← fillnextcover(CI, HidingByteR,G,B) // HidingByte is the Secret image color bytes
  Return SI
}

```

Algorithm 8. is the data hiding algorithm of a secret image SM on N cover images using CipherKey CK to produce N stego images, see Eq. (12).

$$\text{HideImage: SM} \times (\text{CI})^N \times \text{CK} \rightarrow (\text{SI})^N \quad (12)$$

**Algorithm 8. Hide Image**

```

// The main loop on the secret image's pixels
Function HideImage(CustomImage SecretImage, CustomImage(s) CI, String CipherKey)
{
  Let CipherList ← initialize_Cipher(CipherKey) // initialize the random number generator
  For each Vertical pixel Y in SecretImage
    For each Horizontal pixel X in SecretImage
      SI ← DataHidingWrapper(SMX,YHBR,G,B, CI)
  // End of loops
  Save SI // Save the modified cover images, which are now the stego images for extraction.
}

```

Algorithm 8. Shows the primary function used to make a simple loop on all SM pixels and send them to the data-hiding wrapper used by the algorithm. The implementation will be sent to an LSB w/Randomization function, but other data-hiding algorithms from the wrapper can be used instead. Note that the cover images and cipher keys are global variables and do not need to be sent to each function to work. Instead, CI will be modified directly by the process as they are stored in the CustomImages list, and the result will be the stego images of the algorithm.

**4.2 Data Extraction Algorithm**

The extraction algorithm uses the Cipher key and loads the stego images in x load order that matches the original cover images' load order to rebuild the PRNG details and determine the stego image pixels where the secret image data is hidden. The algorithm works similarly to the data-hiding algorithm, except it will extract the least significant bits from the selected random pixels instead of hiding new data in them. The number of extracted bits must match the number used when hiding data.

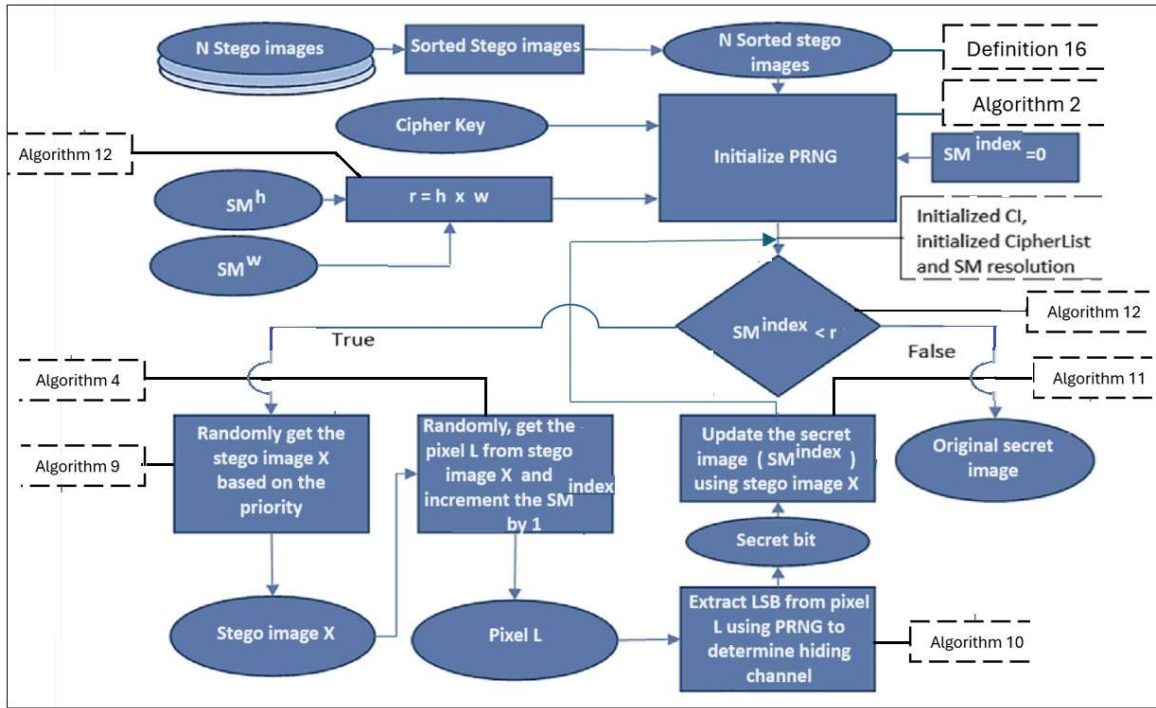


Fig. 3. Data Extraction Architecture

**Definition 14:** Let  $i\_current\_Stego$  be a variable used to keep track of the current stego image index used for data hiding.

**Definition 15:** Let  $i\_inactive\_Stego$  be the number of deactivated stego images; an image is deactivated when the related cover image would have been deactivated during data hiding. When all stego images are present, the user can stop execution or load more cover images.

**Definition 16:** Let  $StegoImages$  be a set of types of Custom images defined in Eq. (13):

$$StegoImages^X \rightarrow \{i \in (0, \infty)\} \quad (13)$$

where  $X$  is the index of a stego image, depending on the order in which the images were loaded. The variable  $x$  starts as 0 for the first loaded cipher image and ends at  $StegoImages^{Length}$ . When all stego images are deactivated, the user can stop execution or load more stego images to utilize a more complex data-hiding pattern.

**Definition 17:**  $PRNG\_NextStegoImg$

Let  $PRNGNextStegoImg$  be the function used to select the next stego image during data extraction, see Eq. (14):

$$PRNGNextStegoImg: i\_current\_stego \rightarrow PRNGTest, \quad \text{where } PRNGTest \in \{true, false\} \quad (14)$$

Before selecting a stego image, the  $nextPixel$  function of the current stego image object is called to perform a test comparing the following random number generated by function in Eq. (14) and the priority of the current stego -image. If the stego image passes this  $PRNGTest$ , the image will be used for data hiding; the function  $PRNGnextStegoImg$  returns true. On the other hand, when a stego -image fails the  $PRNGTest$ , the algorithm starts looking for the next suitable stego image for data hiding. Whenever a stego image fails the  $PRNGTest$ , the function will change the current stego image to the next active one in the stego images list. Once an active stego -image is found, the function will repeat the random number comparison with the current stego image priority to determine if the image should be used again to extract more data. The function recursively loops until an active stego image passes the random number generator test.

Algorithm 9 is a recursive function used to find the next valid Stego image for data extraction. A stego image is invalid if it is inactive or the next value from the random number generator fails the priority comparison during the  $PRNGTest$ . The call to the function  $NextPixel$  contains the implementations for random number generation and priority at a pixel level during data hiding.

**Algorithm 9. Stego image data extraction**

```

Function PRNG_NextStegoImg returns Boolean { // Random cover selection with load balancing
  Let bool continue_Search ← true
  if (StegoImagesinextPixel == false) {
    /* Call customImage nextPixel to test PRNG priority when all stegoimages are used, the algorithm needs to load the next batch of images to
       continue the extraction, similar to how data hiding was performed*/
    if (inactive_Stego = StegoImagescount)
      load more stego images into StegoImages;
    while (continue_Search == true) { // Search for next active CI
      Stegoimagesicurrent_stego ← StegoImagesicurrent_stego+1 mod StegoImagescount
      If StegoImagesiActivecurrent_stego == true then continue_Search ← false
    } // End while loop
    return PRNG_nextStegoImg(); // Recursively call this function to perform nextPixel() PRNG tests
  }
  Return true;
}

```

Algorithm 10 is the function that extracts the SM bits from the next SI that passes PRNG tests and has load-balancing priority.

**Definition 18:** Extract\_Next\_Cover function is defined in the Eq. (15).

$$\text{Extract\_Next\_Cover: } SI_i^R \times SI_i^G \times SI_i^B \rightarrow SI_i^{R'} \times SI_i^{G'} \times SI_i^{B'} \quad (15)$$

Let Extract\_Next\_Cover be the function that calls PRNG\_NextStegoImg defined in Definition 17 to select the next SI and extract the hidden SM bits. Only the LSB of each color channel is retrieved, as the SM will be in those positions.

In addition, Algorithm 10, called the function PRNG\_NextStegoImg, is used to swap to the next active stego image by performing random number and priority tests on the stego images until the next stego image passes the random number checks. After selecting a stego image, the process will

extract the LSB of each color channel. The pseudo-code displayed hides the 3 bits from the secret image into three random pixels in 3 random cover images. PRNG\_NextStegoImg() can be called three times instead of sending each color bit to hide into one specific or random channel. Moreover Algorithm 10 call the function GetPixel(.) Define in Eq.(16) to get one pixel from stego image at the location (x,y).

$$\text{GetPixel: } X \times Y \times SI \rightarrow SI_{x,y}^c, \text{ where } c \in \{R, G, B\} \quad (16)$$

**Algorithm 10. Next cover extraction**

```

Function Extract_next_cover (bit Redoutput, bit greenoutput, bit blueoutput) {
  // All parameters are passed by reference, calling function input parameters will be modified by this function
  Color Extraction_Color // Variable used to retrieve SI pixels data
  if (PRNG_NextStegoImg()) {
    // Extract the pixel from the Stegoimages set at index i_current_stego using the image's HidingX,HidingY position
    Extraction_Color ← StegoImagesiGetPixel(SIHidingX,SIHidingY,SI) //Extract the SI pixel at Hiding positions
    RedOutput ← LSB(ExtractionColorRed) // Extract last bit in red channel
    GreenOutput ← LSB(ExtractionColorGreen) // Extract last bit in Green channel
    BlueOutput ← LSB(ExtractionColorBlue) // Extract last bit in blue channel
  } // Send back the 3 extracted SM bits
  Return (RedOutput, GreenOutput, BlueOutput)
}

```

Algorithm 11 sends bits to the function that selects a random cover image and performs the data-extraction functionality. The dataExtraction\_wrapper works with the Red (R), Green (G), and Blue (B) bytes of a pixel saved in the Color data type, see Eq. (17):

$$\text{DataExtraction\_Wrapper: } SM \times (SI)^N \rightarrow SM_{i,j} \quad (17)$$

where  $SM$  is the secret image,  $(SI)^N$  is the stego images produced by the data hiding, and  $i$  and  $j$  represent the extracted pixel  $x$  and  $y$  positions, respectively.

The function retrieves each bit of the next pixel in the secret message using the function extract\_next\_cover, which will find the next stego image used during data hiding. For example, the three least significant bits from the three-color channels (RGB) will be sent to hide the 3 LSB bits in the following cover images. It is possible to extend this function to do the cover image swap instead of independently for each color channel to store them in different cover images. Still, it increases the total number of calculations.

**Algorithm 11. Data Extraction Wrapper Function**

```

// Extraction wrapper used to call the function that randomly picks a Ci for data hiding
Function DataExtraction_Wrapper (Image SM, StegoImages) return color {
  Let PixelData be of type Color // Color details of next SM Pixel
  Let ba_hide_char_array be a BitArray of length (8)
  Let ba_hide_char_arrayA be a BitArray of length (8)
  Let ba_hide_char_arrayG be a BitArray of length (8)
  Let ba_hide_char_arrayB be a BitArray of length (8)
  Let Bool bitRed ← false
  Let Bool bitGreen ← false
  Let Bool bitBlue ← false
  For I in 1 to 8 // Rebuild each bit of the secret image pixel and rebuild the 8 bits of each color change, 24 bits total
    { (bitred, bitgreen, bitblue) ← extract_next_cover(bitred, bitgreen, bitblue, SM);
      ba_hide_char_array[bit_pos] ← bitred;
      ba_hide_char_arrayG[bit_pos] ← bitgreen;
      ba_hide_char_arrayB[bit_pos] ← bitblue;
    }
  PixelDataI ← bitred * bitgreen * bitblue;
}

```

Algorithm 12 is applied for data extraction at the receiver side.

$$\text{ExtractImage: } H \times W \times CK \times (SI)^N \rightarrow SM \quad (18)$$

**Algorithm 12. Data Extraction Function**

```

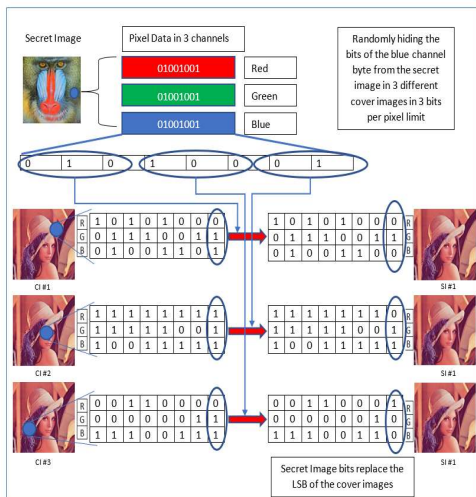
// The main loop on secret image's pixels
Function ExtractImage(Integer Y, Integer X, String CipherKey, CustomImage(s) SI)
{
  initialize_Cipher(CipherKey) // initialize the random number generator
  SecretImage ← new empty image
  For I in 1..Y // loop to rebuild each vertical position in secret image
    For j in 1..x // loop to rebuild each horizontal position in the secret image
      SetPixel(X, Y, SecretImage, Data_Extract_Wrapper(SecretImage, SI)) // Set SMX,Y to extracted CI bits
    // Both loops ended before the return is triggered
  Return SecretImage // Return the secret image built from the extracted data
}

```

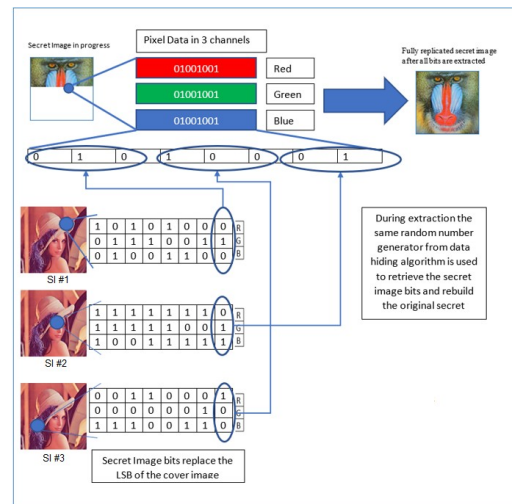
In the algorithm 12, the outlines of the primary function for iterating through all pixels of the secret message (SM) and passing them to the data extraction wrapper. Subsequently, the implementation proceeds to a function that retrieves the bits of the original cover image from stego images, maintaining the same random order used for data concealment. Notably, stego images and cipher keys are global variables, obviating the need for their transmission to each function; instead, CustomImages list is directly modified, yielding the original secret image as the result.

### 4.3 Implementation of the Proposed Hiding and Extraction Algorithm

Fig. 4 and Fig. 5 show the implementation of the data hiding and extraction algorithms on a sample image. During the data hiding algorithm, the data from the blue channel is hidden inside multiple cover images; the same data will be extracted from the same positions during the data extraction algorithm.



**Fig. 4. Data Hiding Implementation**



**Fig. 5. Proposed Data Extraction Implementation**

## 5. Results Discussion

The results of the proposed replication approach have been discussed based on the resolution values achieved and compared with the various steganography algorithms.

### 5.1 Experimental Settings

The algorithm has been implemented at a software application level using Visual Studio 2022 with a C# backend image processing process written from scratch and the randomization and data-hiding algorithms implemented. C# was used because of its object-oriented, bitmap image interface and pointer capabilities. The experimental results were observed using images from the (UCID V2) data set. The tests were performed using the same cover images used by other research, as comparing the results on the same cover image is the way to generate comparable results. The resolution of the cover images used significantly impacts the results; therefore, the cover images must be maintained during comparisons. When comparing multiple smaller cover images with one bigger cover image, all the smaller cover images must match the bigger image's full resolution.

### 5.2 Testing Standard

This section presents the standards for testing the quality of data-hiding algorithms, which will be used later to compare with related works. The following section presents the peak-signal-to-noise ratio (PSNR) and payload capacity (Hardan et al., 2022). The measurement of the quality of data hiding results on a stego-image compared with the cover-image used for data hiding. A higher PSNR means the stego image is closer to the cover image. Thus, higher data hiding quality is more challenging to detect as a modified image. Presents analysis on PSNR, including the following formula to calculate PSNR:

$$\text{PSNR}(CI, SI) = 10 \times \log_{10} \left( \frac{\text{Max}(255)^2}{\text{MSE}(CI, SI)} \right) \quad (19)$$

Meanwhile, mean square error (MSE) measures the average differences between two images defined in Eq. (20).

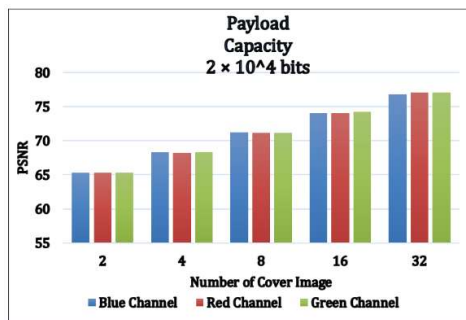
$$\text{MSE}(CI, SI) = \frac{1}{h \times w} \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} (CI(i, j) - SI(i, j))^2 \quad (20)$$

Because MSE works at a per-pixel level, PSNR calculates the overall similarity between the original cover image (CI) and its modified form in a stego-image (SI). The payload defined in Eq. (21) equals the number of bits hidden in the cover image (s) to create stego-images. Higher payloads increase MSE and reduce PSNR, making it easier to detect an image as a possibly modified cover image.

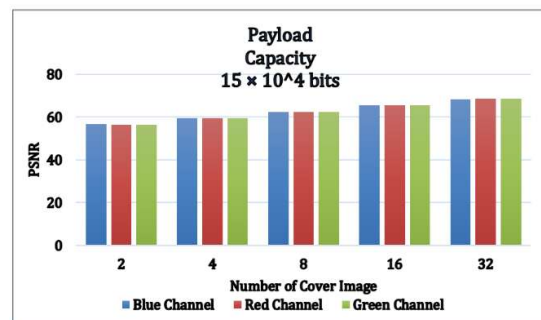
$$\text{Payload} = \left\lfloor \text{SM}_{\text{Width} \times \text{Height} \times 3 \times 8} \right\rfloor \quad (21)$$

### 5.3 Effect of Replication Cover Images on the Security Level

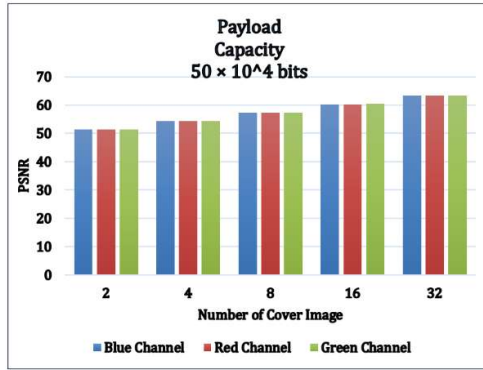
The PSNR metric was used to check the imperceptibility of images concerning payload capacity. Figs. (6, 7, 8, and 9) illustrate the effect of employing multiple cover images ( $256 \times 256$ ) resolution to conceal different payload capacity data by utilizing the three channels, which enhances PSNR by 9% on average. Moreover, maximum PSNR is found when the number of cover images gradually increases in any payload capacity. Hence, the maximum PSNR is found when the number of cover images is 32 at the payload capacity is  $2 \times 10^4$  at the red channel. In contrast, the minimum PSNR is (51.34 dB) when the two cover images have a payload capacity of  $50 \times 10^4$  at the blue channel. In addition, the maximum variance of three channels is found when we used 32 cover images, 16 cover images, and two cover images at the payload capacity  $2 \times 10^4$ ,  $15 \times 10^4$ , and  $50 \times 10^4$ , respectively.



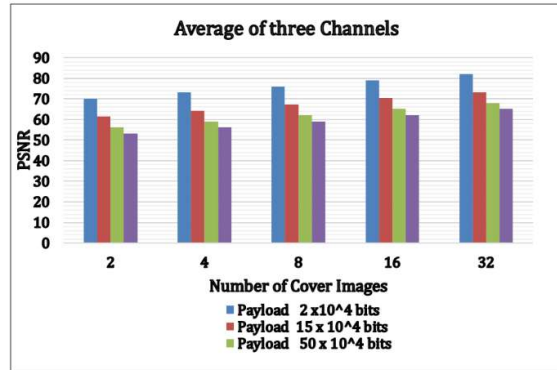
**Fig. 6.** The PSNR of data hiding in cover images ( $256 \times 256$ ) on three channels with the load capacity ( $2 \times 10^4$  bits)



**Fig. 7.** The PSNR of data hiding in cover images ( $256 \times 256$ ) on three channels with the load capacity ( $15 \times 10^4$  bits)



**Fig. 8.** The PSNR of data hiding in cover images (256×256) on three channels with the load capacity (50×10<sup>4</sup> bits)



**Fig. 9.** The average PSNR of data hiding in cover images (256×256) on three channels using several load capacities

#### 5.4 Result in Comparison with previous work

Image imperceptibility comparisons were conducted to evaluate the proposed algorithm's performance compared to existing methods. The focus was on assessing the imperceptibility of stego images generated by the algorithm when using single cover images, as this aspect is crucial for ensuring the visual integrity of the resulting images. The evaluation included comparisons with algorithms prioritizing achieving higher PSNR values. As these algorithms typically operate on single cover images, they serve as a benchmark for assessing the effectiveness of our approach. As we mentioned before, by gradually increasing the number of cover images used in the embedding process, the PSNR values of resulting stego images were observed to rise accordingly. This observation underscores the importance of finding the optimal balance between the number of cover images utilized and the resulting PSNR values. Identifying the threshold where employing multiple cover images yields comparable or superior results in terms of PSNR represents a significant milestone. This finding indicates our approach's potential to achieve high-quality stego images while leveraging the benefits of multiple cover images. The results of the imperceptibility comparisons demonstrate the effectiveness of the proposed algorithm in generating stego images with enhanced visual quality, even when utilizing various cover images.

**Table 1**

PSNR Comparison with Related Works


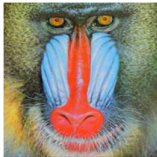

Name of CI used	Payload Capacity Bits × 10 <sup>4</sup>	PSNR (dB) using Ou, B., et al., (2015)	PSNR (dB) using Li, J., (2013)	PSNR (dB) using MDLSB Elshare, S., EL-Emam, N. (2020)	PSNR (dB) using IDHA Ahmad, M. et al., (2021)	PSNR (dB) using the proposed data hiding on a single image
Lena 	2	60.6	59.1	62.7	65.8	67.1
	6	55.3	53.8	57.3	60.1	62.3
	8	53.8	52.5	55.6	58.3	61.0
	12	51.2	50.4	54.8	57.5	59.3
	15	49.6	N/A	53.1	55.7	58.3
Baboon 	2	56.8	57.1	59.2	63.3	67.0
	2.8	54.9	55.2	56.1	57.8	65.6
	3.6	53.3	53.3	54.9	56.1	64.56
	4.4	51.9	51.9	53.2	55.4	63.64
	5.6	49.8	49.9	53.4	53.6	62.61
Barbara 	2	62	59.8	63.2	65.1	66.95
	5	57	55.8	59.6	59.4	63.03
	7	55.1	54.1	58.5	57.7	61.62
	10	53	52.5	55.9	56.9	60.10
	12.5	51.1	51.1	54.8	55.1	59.15

Table 1 and Figs. (10-12) outline the payload capacity and PSNR of cover images with various testing algorithms, with the current research algorithm concealing data in a single cover image. The results indicate higher PSNR across all cases than previous works (Ou et al., 2015; Li, 2013; Elshare & EL-Emam, 2020; Ahmad et al., 2021). The minimum differences in PSNR were observed between the proposed work and Ahmad, M. et al., (2021) for all payload capacities and all images. The results show varying ranges from 1.3 to 2.6 in the Lena image and from 3.7 to 9.014 in the Baboon image, with differences ranging from 1.85 to 4.05 in the Barbara image.

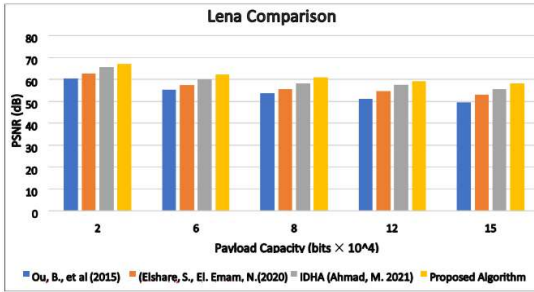


Fig. 10. Lena Image Distortion Comparison to Related Works

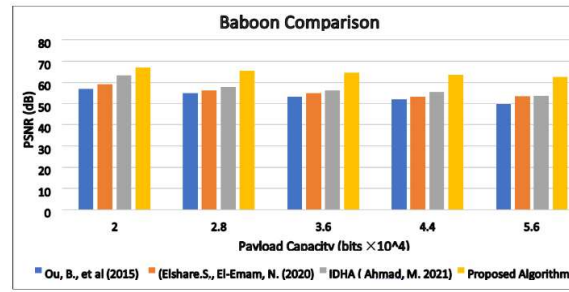


Fig. 11. Baboon Image Distortion Comparison to Related Works

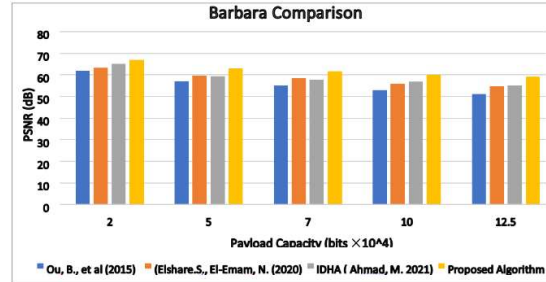


Fig. 12. Barbara Image Distortion Comparison to Related Works

Table 2

Demonstrates the impact of using fewer Lena cover images than a larger number

Payload Capacity Bits × 10 <sup>4</sup>	Number of 256×256 cover-images used in the proposed work	Lowest PSNR of Stego-Images in the proposed work	Average PSNR of Stego-Images in the proposed work	PSNR in one 512×512 image Using IDHA (Ahmad, M. 2021)
2	1	61.1	61.1	65.8
	2	64.06	64.11	65.8
	3	65.84	65.85	65.8
	4	67.06	67.12	65.8
8	1	55.05	55.05	58.3
	2	58.03	58.03	58.3
	3	59.76	59.80	58.3
	4	61.07	61.08	58.3
15	1	52.33	52.33	55.7
	2	55.30	55.31	55.7
	3	57.06	57.07	55.7
	4	58.30	58.32	55.7

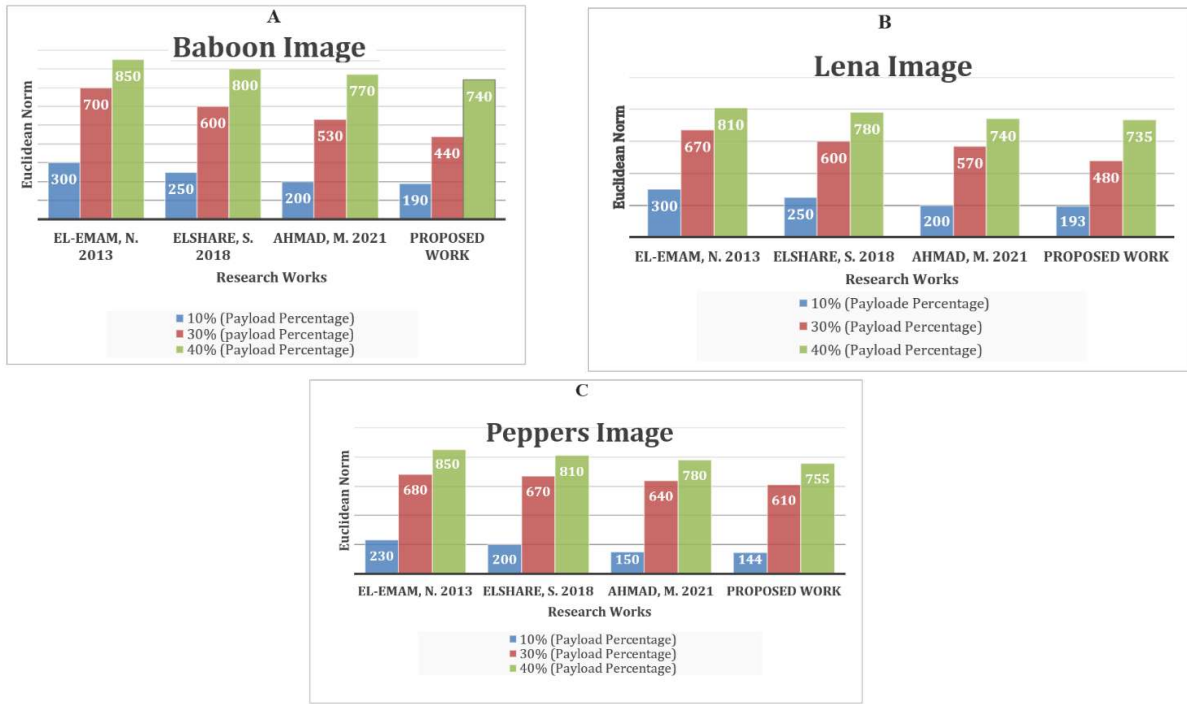
Table 2 shows the impact of all of Lena's features on maintaining PSNR relevance. Four 256×256 images achieve slightly higher PSNR than one 512×512 image with the same total resolution. Data hiding involved 3 bits per pixel in each byte. The results indicate that the PSNR of three smaller 256×256 images closely match that of one larger 512×512 image. Additionally, the proposed algorithm outperforms a single larger image by 2.62% at a payload capacity of 15×10<sup>4</sup> bits.

### 5.5 Euclidean norm

Euclidean norm test Eq. (22) has been implemented on three-color images with the size (512 × 512) to check the distance (d) between cover-image and stego-image for three color components {R, G, B}.

$$d = \sqrt{(R_{CI} - R_{SI})^2 + (G_{CI} - G_{SI})^2 + (B_{CI} - B_{SI})^2} \quad (22)$$

The smallest distance will be reached when the proposed algorithm is implemented. Furthermore, the results indicate that the greatest variance between the distance of the proposed algorithm and previous works occurred in (El-Emam, 2013), with a difference of 260 for the Baboon image at a 30% payload percentage. Additionally, the greatest difference between the distance of the proposed algorithm and previous works is evident in (Ahmed et al., 2022), with a difference of six for the Peppers image at a 10% payload percentage. See Figs. 13 (A, B, C).



**Fig. 13(A-C).** The effects of Euclidean norms vs payload capacity for the proposed algorithm and other research works

### 5.6 Avoiding WFLoSv attack

The main intention of the suggested hiding algorithm is concealing a secret message  $S_m$  in the color image without making suspicion that the stego-image contains hidden information. In this work, the stego-image produced by the suggested algorithm IDHA has been examined against the WFLoSv attacker using the “Receiver Operating Characteristic” (ROC) curve (Elshare & EL-Emam, 2020). The ROC curve is based on two parameters: the probability of false alarms ( $P_{FA}$ ) and the probability of detection ( $1 - P_{MD}$ ), see Eqs. (23, 24).

$$P_{FA} = \frac{NCI(SI)}{NCI} \quad (23)$$

$$P_{MD} = \frac{NSI(CI)}{NSI} \quad (24)$$

where

$P_{FA}, P_{MD} \in [0,1]$ ,  $NCI(SI)$  is several cover images recognized as stego-images,  $NCI$  is the total number of cover images, and  $NSI(CI)$  is the number of stego-images recognized as cover images.  $NSI$  is the total number of stego-images. In the ROC scheme,  $P_{FA}$  is presented on the horizontal axis, while  $(1 - P_{MD})$  is presented on the vertical axis. A steganography technique is said to be secure from an attacker if the following condition is satisfied see Eq.(25):

$$|P_{FA} - 1 + P_{MD}| = \varepsilon, \text{ and } \varepsilon \rightarrow 0 \quad (25)$$

It means that the area under a curve  $AUC=0.5$ , whereas the perfect detection of attackers is found when  $\varepsilon \rightarrow 1$ ; see (Hardan, 2022).

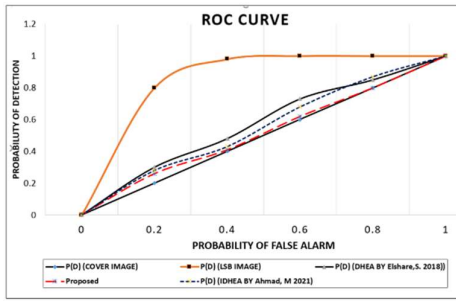
### 5.7 ROC curve

A ROC curve (receiver operating characteristic curve) is a graph that illustrates how well a classification model performs at various classification thresholds. This curve represents two parameters:

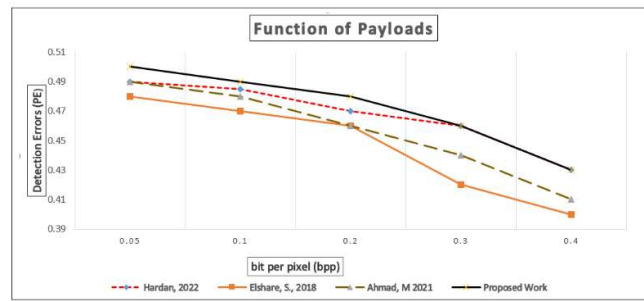
- True Positive Rate (TPR), which is synonymous with recall and is defined as  $TP / (TP + FN)$
- False Positive Rate (FPR), which is defined as  $FP / (FP + TN)$



The ROC curve plots TPR against FPR at different classification thresholds. When the classification threshold is lowered, more items are classified as positive, increasing both False Positives and True Positives. The figure below demonstrates a typical ROC curve.



**Fig. 14.** ROC curves of the WFLogSv steganalyser against the proposed hiding algorithm MDLSB and the traditional LSB with a payload of 40% capacity



**Fig. 15.** The dissimilarity between adjacent pixels with payload is 40%

In Fig. 14, the performance of this study is compared with that of other researchers in defending against the WFLogSv attack using 40% of payload capacity. The study observed that the WFLogSv attack shows strong detection capabilities when traditional LSB is applied but weak detection when our approach is used. The detection rate for identifying a secret message using the proposed technique does not exceed 2.9%, whereas the detection rates using the DHEA technique (Elshare & EL-Emam, 2020) and the IDHEA technique (Ahmed et al., 2022) are approximately 16.7% and 14.0%, respectively. Moreover, the proposed technique has better detection rates than the previous works mentioned above, at approximately 13.8% and 11.16%, respectively.

### 5.7 Estimate the detection error ( $P_E$ )

The results confirm that the suggested hiding algorithm with a replication technique is highly imperceptible and works against attacks for different payload capacities (see Fig. 15). Furthermore, the performance of the present steganography can be achieved by detection error ( $P_E$ ) expressed in Eq. (26).

$$P_E = \min \left( \frac{1}{2} (P_{FA} + P_{MD}) \right) \quad (26)$$

The error  $P_E$  lies in the range  $[0, 0.5]$ , where zero corresponds to perfect detection and (0.5) to perfect security of the steganographic scheme Figure 15. Detection error is estimated as a function of payload capacity based on bit per pixel (bpp) to find the area under the curve (AUC). The result of the suggested algorithm has been analyzed and compared with the previous work (Elshare, & EL-Emam, 2020; Ahmed et al., 2022; Hardan et al., 2022). The proposed replication algorithm results show that the mean value of PE equals 0.472. This result is better than the previous works mentioned above by about 4.8%, 2.8%, and 0.6%, respectively. Moreover, the excellent security percentage reaches 99% when  $\text{bpp} = 0.05$ , whereas the worst security percentage reaches 86% when  $\text{bpp} = 0.4$ .

## 6. Conclusion and future research

This paper proposes a new steganography algorithm to enhance attack protection based on a replication technique. The PSNR is effectively applied to produce the new data-hiding algorithm for color images. The search used randomized LSB data hidden across multiple cover images to enhance security. A complex pseudo-random generation makes it easier for intended recipients to recover data and makes it difficult for attackers to uncover patterns within stego-images. Loading stego-images in multiple batches increases brute force complexity while using various images reduces distortion compared to a larger image. The new algorithm allows more flexible data hidden across different cover images simultaneously. The results show that the best PSNR, payload capacity, and Euclidean norm are better than the previous works, and the attack resistance is perfect.

## References

- Abuali, M. S., Rashidi, C. B. M., Raof, R. A. A., Azir, K. N. F. K., Hussein, S. S., & Abd-Alhasan, A. Q. (2024). Enhancing Security with Multi-level Steganography: A Dynamic Least Significant Bit and Wavelet-Based Approach. *Mathematical Modelling and Engineering Problems/Mathematical Modelling of Engineering Problems*, 11(6), 1403–1416. <https://doi.org/10.18280/mmep.110602>

- Ahmad, M., El-Emam, N. N., & Al-Azawi, A. F. (2021). Improved Deep Hiding/Extraction Algorithm to Enhance the Payload Capacity and Security Level of Hidden Information. *International Journal of Communication Networks and Information Security*, 13(3). <https://doi.org/10.17762/ijcnis.v13i3.4759>
- Alabaichi, A., Al-Dabbas, M. a. a. K., & Salih, A. (2020). Image steganography using the least significant bit and secret map techniques. *International Journal of Electrical and Computer Engineering*, 10(1), 935. <https://doi.org/10.11591/ijece.v10i1.pp935-946>
- Al-Shaaby, A. A., & AlKharobi, T. (2017). Cryptography and Steganography: New Approach. *Transactions on Networks and Communications*, 5(6). <https://doi.org/10.14738/tnc.56.3914>
- Ansari, A. S., Mohammadi, M. S., & Parvez, M. T. (2020). A Multiple-Format Steganography Algorithm for Color Images. *IEEE Access*, 8, 83926–83939. <https://doi.org/10.1109/access.2020.2991130>
- Baluja, S. (2017). Hiding Images in Plain Sight: Deep Steganography. <https://api.semanticscholar.org/CorpusID:29764034>
- Elharrouss, O., Almaadeed, N., & Al-Maadeed, S. (2020). An image steganography approach based on k-least significant bits (k-LSB). *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*. <https://doi.org/10.1109/iciot48696.2020.9089566>
- Elshare, S., & EL.Emam, N. N. E. (2020). Modified multi-level steganography to enhance data security. *International Journal of Communication Networks and Information Security*, 10(3). <https://doi.org/10.17762/ijcnis.v10i3.3614>
- Haider, T., Blanco, S. A., & Hayat, U. (2024). A novel pseudo-random number generator based on multivariable optimization for image-cryptographic applications. *Expert Systems With Applications*, 240, 122446. <https://doi.org/10.1016/j.eswa.2023.122446>
- Hardan, H., Alawneh, A., & El-Emam, N. N. (2022). New deep data hiding and extraction algorithm using multi-channel with multi-level to improve data security and payload capacity. *PeerJ. Computer Science*, 8, e1115. <https://doi.org/10.7717/peerj-cs.1115>
- Li, J., Li, X., & Yang, B. (2013). Reversible data hiding scheme for color image based on prediction-error expansion and cross-channel correlation. *Signal Processing*, 93(9), 2748–2758. <https://doi.org/10.1016/j.sigpro.2013.01.020>
- Liu, X., Ma, Z., Ma, J., Zhang, J., Schaefer, G., & Fang, H. (2022). Image Disentanglement Autoencoder for Steganography without Embedding. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. <https://doi.org/10.1109/cvpr52688.2022.00234>
- Nie, S. A., Sulong, G., Ali, R., & Abel, A. (2019). The use of Least Significant Bit (LSB) and Knight Tour Algorithm for image steganography of cover image. *International Journal of Electrical and Computer Engineering*, 9(6), 5218. <https://doi.org/10.11591/ijece.v9i6.pp5218-5226>
- Ou, B., Li, X., Zhao, Y., & Ni, R. (2015). Efficient color image reversible data hiding based on channel-dependent payload partition and adaptive embedding. *Signal Processing*, 108, 642–657. <https://doi.org/10.1016/j.sigpro.2014.10.012>
- Pei, Y., Luo, X., Zhang, Y., & Zhu, L. (2020). Multiple Images Steganography of JPEG Images Based on Optimal Payload Distribution. *Computer Modeling in Engineering & Sciences*, 125(1), 417–436. <https://doi.org/10.32604/cmescs.2020.010636>
- Plachta, M., Krzemiński, M., Szczypiorski, K., & Janicki, A. (2022). Detection of Image Steganography Using Deep Learning and Ensemble Classifiers. *Electronics*, 11(10), 1565. <https://doi.org/10.3390/electronics11101565>
- Sun, Y., Zhang, H., Zhang, T., & Wang, R. (2019). Deep neural networks for efficient steganographic payload location. *Journal of Real-time Image Processing*, 16(3), 635–647. <https://doi.org/10.1007/s11554-019-00849-y>
- Wang, J., Yang, C., Wang, P., Song, X., & Lu, J. (2020). Payload location for JPEG image steganography based on co-frequency sub-image filtering. *International Journal of Distributed Sensor Networks*, 16(1), 155014771989956. <https://doi.org/10.1177/1550147719899569>
- Wei, P., Li, S., Zhang, X., Luo, G., Qian, Z., & Zhou, Q. (2022). Generative Steganography Network. Proceedings of the 30th ACM International Conference on Multimedia. <https://doi.org/10.1145/3503161.3548217>
- Yang, C., Liu, F., Ge, S., Lu, J., & Huang, J. (2019). Locating secret messages based on quantitative steganalysis. *Mathematical Biosciences and Engineering*, 16(5), 4908–4922. <https://doi.org/10.3934/mbe.2019247>
- Zhu, X., Lai, Z., Zhou, N., & Wu, J. (2022). Steganography with High Reconstruction Robustness: Hiding of Encrypted Secret Images. *Mathematics*, 10(16), 2934. <https://doi.org/10.3390/math10162934>

