Price Hiller
Kevin Pham
Joseph Baca
Kaitlyn Grace
Project Group 2

A Novel Data Embedding Method in RGB Color Component Bit Planes

Steganography CS-4643-01T
August 7th, 2025

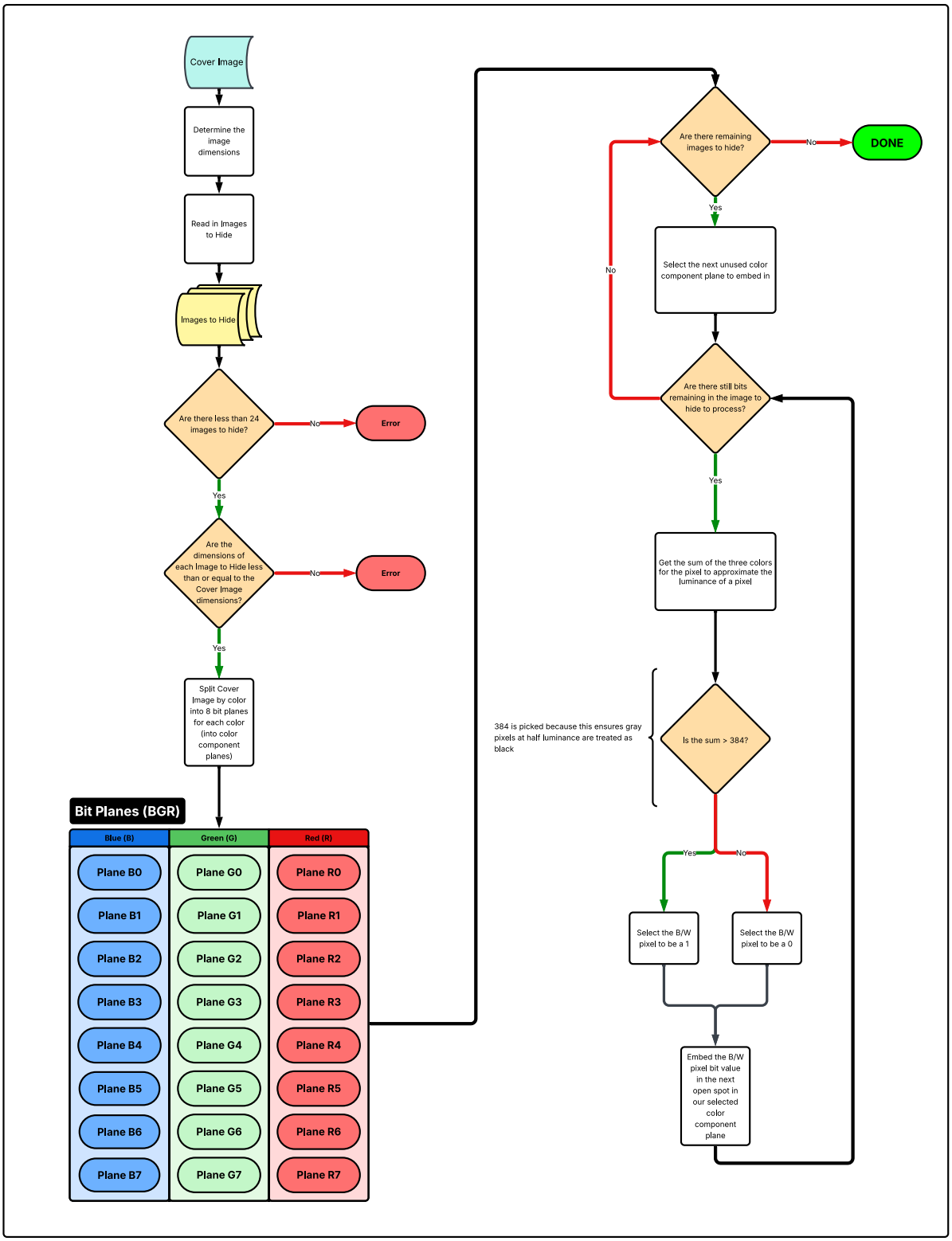
Introduction

In this project, we developed a program which embeds multiple images within the individual bit planes of each color component in a 24-bit color bitmap. To accomplish this, the program converts input images to black and white and then embeds each image in a single color component bit plane by distributing its bits across the plane linearly. This means treating each of the red, green, and blue components as individual bit planes [1]. This allows up to 23 images to be hidden if we preserve a single color component bit plane of the cover image. The goal of this project was to implement a novel technique for concealing multiple images within a single bitmap file, maximizing hiding capacity.

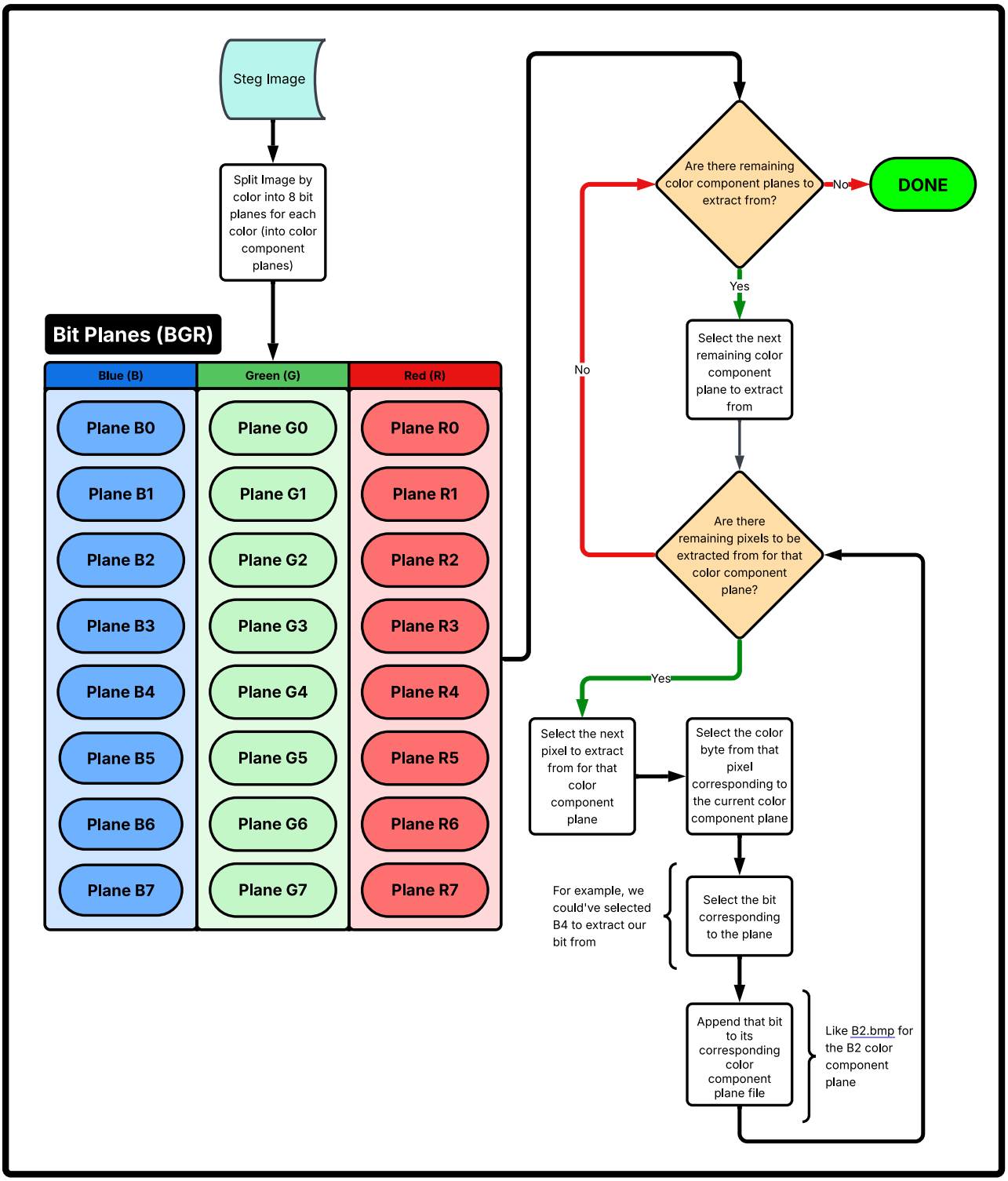
We conducted an analysis on the output to determine the effectiveness of our method, including the visual detectability of the embedded data. This analysis will also discuss the techniques involved with this method, difficulties we faced in development, suggestions for improvement in the future, as well as examples of the finished product.

Functional Block Diagram

Hiding



Extraction


Steg Image
Split Image by color into 8 bit planes for each color (into color component planes)
Bit Planes (BGR)
Blue (B)
Green (G)
Red (R)
Plane B0
Plane B1
Plane B2
Plane B3
Plane B4
Plane B5
Plane B6
Plane B7
Plane G0
Plane G1
Plane G2
Plane G3
Plane G4
Plane G5
Plane G6
Plane G7
Plane R0
Plane R1
Plane R2
Plane R3
Plane R4
Plane R5
Plane R6
Plane R7
Are there remaining color component planes to extract from?
No
DONE
Yes
Select the next remaining color component plane to extract from
Are there remaining pixels to be extracted from for that color component plane?
No
Yes
Select the next pixel to extract from for that color component plane
Select the color byte from that pixel corresponding to the current color component plane
Select the bit corresponding to the plane
For example, we could've selected B4 to extract our bit from
Append that bit to its corresponding color component plane file
Like B2.bmp for the B2 color component plane

Algorithm: Hiding & Extraction

Hiding

The main method of hiding in this project is using similar patterns as Least Significant Bit (LSB) bit-plane-based Steganography [2]. It takes each input image to hide, converts it to a black-and-white image (another technique described below), and embeds it within a individual bit-plane of an unused color component in the 24-bit color bitmap cover image. The target bit within the bit-plane of the chosen color channel is replaced with a bit from one of the black-and-white images. This is repeated until the black-and-white images are completely embedded. This allows up to 23 images to be embedded within the cover image as we reserve the most significant green color component bit plane of the cover image.

Another technique used in this program is to use threshold-based conversion on the pixels within the images to be hidden. The color component value of each pixel is summed and then compared against a threshold to determine if that pixel qualifies as a black or white pixel in the converted black-and-white embedded image. The pixel's summed value must be more than 384 to qualify as a white pixel, otherwise it will embed as a black pixel. Once this finishes, the image that is embedded into the cover is entirely black-and-white and is placed across a single color bit plane as discussed above.

A micro-optimization made to slightly reduce perceptibility in the implemented program is the reservation of the most significant green color component bit plane as discussed previously. The human visual system perceives green strongest and thus the most impactful green colors in the image are preserved from the cover in an attempt to improve security. [3]

Extraction

Extraction is much more straightforward. Unless like normal bit plane extraction, we slice the image up into 24 separate planes, 8 color bit planes per color component. We then extract each individual color component bit plane as a single black and white image where a bit value of 0 is black and a bit value of 1 is white. We then append each bit into its corresponding color component plane file, e.g. `B2.bmp` for the 2nd bit plane in the blue component.

Technical Difficulties

Performance

One of the primary challenges that we encountered in the development of this program was performance. Initially, it took nearly or over twenty seconds to embed and extract a single bit-plane, which made for an inefficient implementation.

Writing the Image Data

Originally, the implementation of this program involved passing full black-and-white byte values to the Python Imaging Library (`PIL`), with each bit representing an individual pixel. However, this led to distorted output images. This meant the image data had to be written back to the disk in the correct format. It took several hours of trial and error to find that feeding individual bits rather than grouped byte values outputs correct results.

Dependency Management

Writing in Python proved to have its own challenges. Dependency management within Python's ecosystem turned out to be frustrating and time-consuming.

Black-and-White Images

Determining the threshold for what makes a black or white pixel when converting images to be hidden was yet another challenge we faced in the development of this project. The original threshold of 383 left pixels with exactly half luminosity as white pixels, which made existing histogram images lose their inner gray bars when embedded and extracted. Raising the threshold to 384 caused the pixels to be rendered as black instead of white, which improved the histograms we embedded.

Suggestions for Future Work

A possible future feature could be implementing an algorithm that changes the order of the images to preserve the most cover image data as possible in order to decrease the visual and possibly data analytical perceptibility. Another way to decrease visual perceptibility would be to encode our data in a spiral as well as arrange our data in a way that it weights the amount of images hidden in each color channel based on the color composition of the color image to account for human visual perception [3].

Currently our histograms for our cover images containing higher quantities of hidden images have a significant spike at 0. Determining an improved algorithm to convert images to black and white, or possibly determining a system to invert some of the black and white embedded images could reduce the 0 spike in the histogram. Additionally, we could analyze the encoded image with visual metrics as well, such as SSIM, PSNR and MSE to evaluate the perceptibility of the modifications in our steganographic images.

Example Images With Data Hidden

Sunset Tree

Image Set

Figure 1: Sunset Tree Original Image

Figure 2: Sunset Tree With 5 Images Embedded

Figure 3: Sunset Tree With 9 Images Embedded

Figure 4: Sunset Tree With 14 Images Embedded

Figure 5: Sunset Tree With 20 Images Embedded

Figure 6: Sunset Tree With 23 (Max) Images Embedded

Histograms

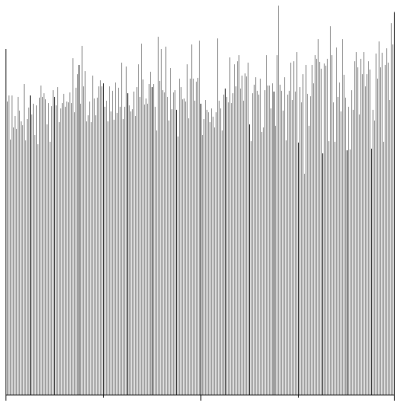


Figure 7: Sunset Tree Original Image

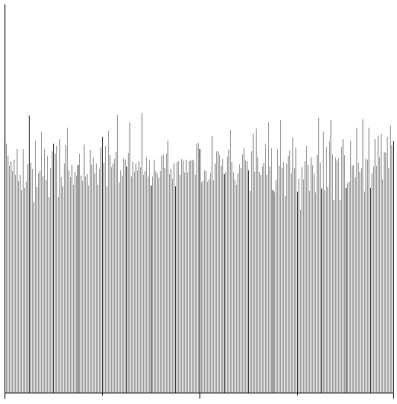


Figure 8: Sunset Tree With 5 Images Embedded

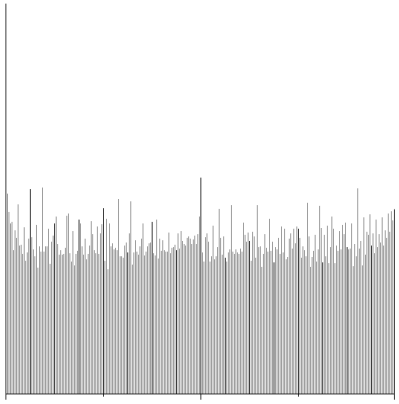


Figure 9: Sunset Tree With 9 Images Embedded

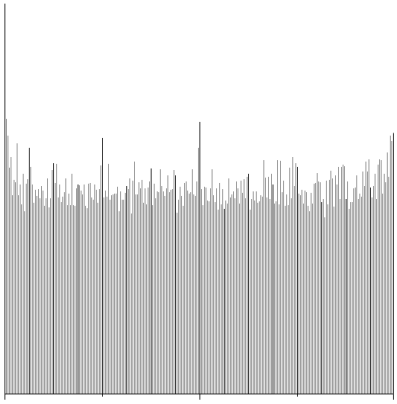


Figure 10: Sunset Tree With 14 Images Embedded

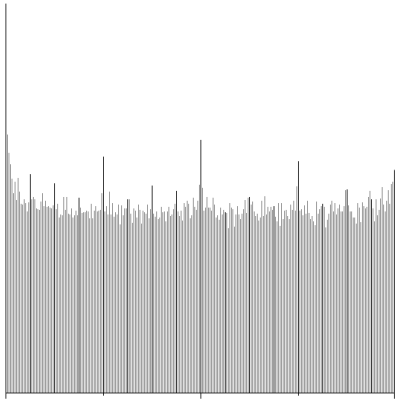


Figure 11: Sunset Tree With 20 Images Embedded

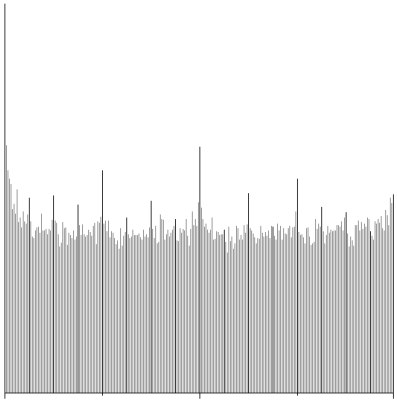


Figure 12: Sunset Tree With 23 Images Embedded

Mantis

Image Set

Figure 13: Mantis Original Image

Figure 14: Mantis With 4 Images Embedded

Figure 15: Mantis With 8 Images Embedded

Figure 16: Mantis With 12 Images Embedded

Figure 17: Mantis With 20 Images Embedded

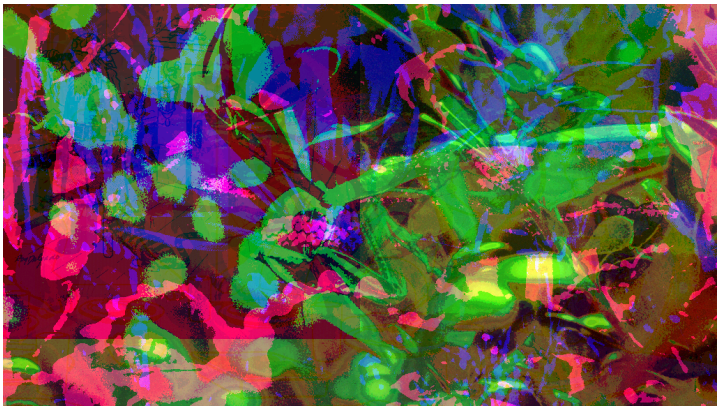


Figure 18: Mantis With 23 (Max) Images Embedded

Histograms

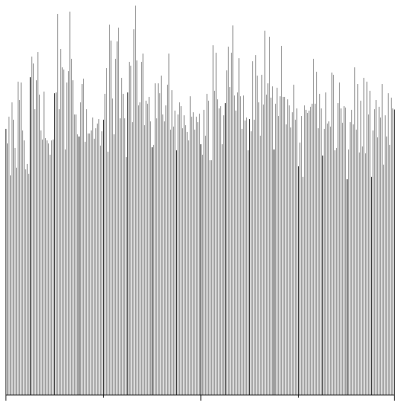


Figure 19: Mantis Original Image

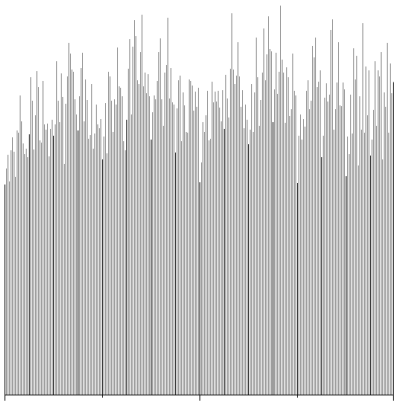


Figure 20: Mantis With 4 Images Embedded

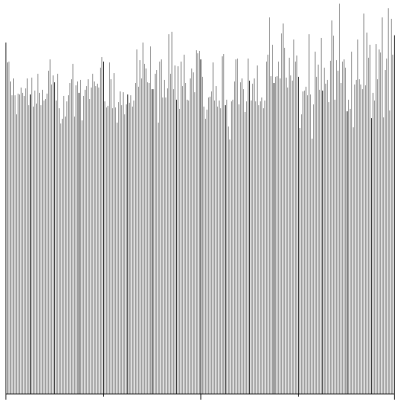


Figure 21: Mantis With 8 Images Embedded

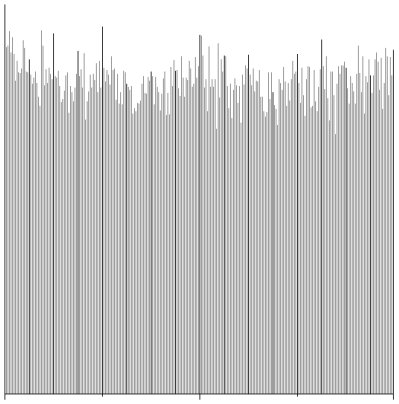


Figure 22: Mantis With 12 Images Embedded

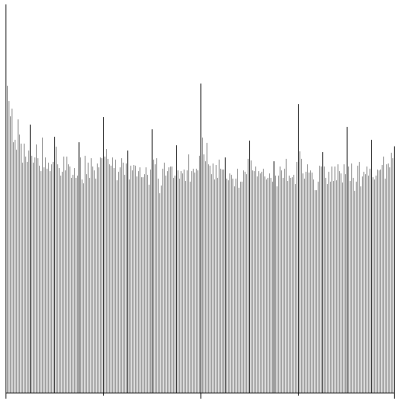


Figure 23: Mantis With 20 Images Embedded

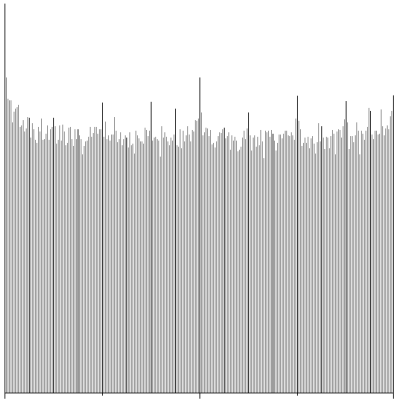


Figure 24: Mantis With 23 Images Embedded

Some Sample Embedded & Extracted Images

Figure 25: Original Pirate Image

Figure 26: Extracted Pirate Image from Mantis

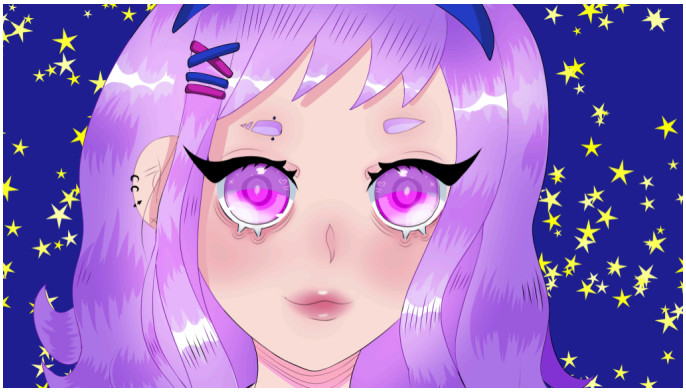


Figure 27: Original Anime Image

Figure 28: Extracted Anime Image from Sunset Tree

Statistical Analysis Results

Our hiding technique allowed for a significantly higher capacity of images to be embedded within a single 24-bit cover image, permitting up to 23 images to be embedded within the cover. Capacity in our technique is best understood not in terms of the number of bits or bytes embedded; rather, going by how many images can be embedded is what is relevant. So long as the images to be hidden have dimensions that are less than the cover, they can be embedded. Thus, the capacity is 23 correctly sized images. To understand it as a ratio then, if we had images we were to embed that all matched dimensions of the cover image, we could embed into 95.8% of the cover image's size in bits.

$$23 \text{ available planes to embed in} \div 24 \text{ total planes} \approx 0.958$$

Our technique had a fairly low perceptibility rate, with cover images subjectively exceeding acceptable perceptibility when around 14 of the 23 available planes to embed in were used. This varied based on the complexity of the image of course. If you review the earlier images in this paper, the Mantis embeddings had little to no perceptibility at around 12 images; whereas the Sunset Tree samples were exceeding perceptibility thresholds around 9 images embedded.

Considering the histograms of the samples, this technique clearly fails to pass histogram analysis even with only a handful of images embedded. Large peaks began to build early in the histogram at low values, likely as a result of the black-and-white conversion process leading to larger quantities of straight 1s and 0s across planes. However, histogram analysis did not reveal absolutely blatant plateaus and cliffs across the histogram like normal LSB techniques would create. With additional methods or embedding regimens to smooth out the early peaks, this embedding technique has a fairly strong chance of bypassing cursory histogram analysis on avoiding cliffs and plateaus alone.

The other primary challenge that this technique faces in bypassing histogram analysis is how much it smooths down the remaining portions of a histogram. Evaluating the histograms embedding from only a handful of images up to even around 12 images embedded, there's a flattening of all regions of the histogram. This flattening becomes more pronounced the more images embedded and reacts strongly to the images embedded in the cover. If we embed a image made of large sections of similar colors, it's likely they'll get converted into large regions of white and black in the embedded image causing long runs of 1s or 0s to appear in the cover when embedded. Superior black-and-white conversions or even permutation/ interleaving of the bits after conversion could reduce the amount of flattening.

Bibliography

[1] A. Singh and H. Singh, "An improved LSB based image steganography technique for RGB images," *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, India*, pp. 1–4, May 2015, doi: 10.1109/ICECCT.2015.7226122.

[2] R. Dumre and A. Dave, "Exploring LSB Steganography Possibilities in RGB Images," *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kharagpur, India*, pp. 1–7, Jul. 2021, doi: 10.1109/ICCCNT51525.2021.9579588.

[3] A. AbdelRaouf, "A new data hiding approach for image steganography based on visual color sensitivity," *Multimedia Tools and Applications*, vol. 80, pp. 23393–23417, Jan. 2021, doi: 10.1007/s11042-020-10224-w.