RGB Herring

Matthew Sutton 45629020, Calvin Crino 92894198, James Bors 04124019 , Hopefully a 4th person

RGB Herring is a program for the steganographic encoding of text into bmp images. RGB Herring supports multiple encoding schemes with variable levels of detectability, and message density. The most basic encoding available uses the least significant bits of the red green and blue channels of the bitmap, to store the message. Additional encoding schemes like, monochrome and keyed, vary the method by which the LSB's are used to add higher read complexity and increased security to decode attacks.

Comparison of principle methods:

(D, Density is how many bits stored per pixel)
(E, Entropy is the amount of changed bits per pixel, worst case)
(S, Security is an amount of tries required to decode given knowledge of method of encoding, detectability is harder to quantify)
(M, Message length in bits)
(I, Image size in pixels)
(KR, Key Range, the amount of valid keys that generate unique pseudo-random sequences, depending on features of rand, $\sim 2^{32}$)

- LSB hijack monochrome - D:1,E:1,S:3
- LSB hijack chromatic keyed - D:3,E:3,S:min(6*KR, $6^{M/3}$)

Method Descriptions:

LSB hijack monochrome:

Uses the last bit of every red green or blue channel to store the message

LSB hijack chromatic keyed:

Uses a pseudo-random number generator to determine which permutation of rgb is the order in which the last bit of the channels is the order of the bits in the message, key provided by user

Milestones:

1) We have to set up a nice UI / I/O for the user. Input of key / image file / message / etc.
2) General picture reading: input picture, output picture, make sure the picture can still be opened by other programs
3) Having a loop change every set amount of R/B/G elements. (LSB hijack monochrome). In this step we can make sure everything is working as intended before randomness takes over.

4) Outputting the now changed picture so that a program can open it. This might be harder than it sounds. We have to make sure none of the encoding gets messed up by our code.
5) Then we get into the random changing of the elements. Do we change only R or B or G, or whatever element is the next random one that pops up. Probably the latter (if we actually cared about the "safety" of our message)
6) Debugging, optimization, making sure that the encoding works for the extremes.

Proposed schedule

10/01 - assignment date - We start from here.
10/02 - 8 weeks - I/O and UI (there isn't extensive UI options in MARS really just dialog boxes)
10/09 - 7 weeks - Conversion between ascii string and bit stream and vice versa (edian-ness)
10/16 - 6 weeks - Byte Array
10/23 - 5 weeks - LSB hijack monochrome implemented
10/30 - 4 weeks - LSB hijack chromatic keyed being worked on, if progress allows, more elaborate encoding implemented.
11/06 - 3 weeks - LSB hijack chromatic keyed finished being implemented, report started
11/13 - 2 weeks - Focusing on the report

Collaboration Summary

1. Calvin - I/O set up, decrypting the image, picture input to byte array.
2. Matthew - LSB bit manipulation to encoded message.
3. James - Conversion from ascii string to bitstream, breaking down image into an array of bytes than can be used by Matthew to encode.
4. Together - Collaborate on the report discussions

Problems we may encounter

- The image that our program returns looks too different from the one that it receives.
    - The more bits are changed, the worse the pixel clarity is throughout the image.
    - We can suppress these strange looking pixels by only changing the least significant bit on any given pixel, ensuring that the color will not be far off from the original.
- Transferring an image into a byte array could be very challenging.
    - Mostly confirming our algorithm is correct.
- Incorrectly decoding the message.
    - Mostly confirming our algorithm is correct.