Steganography is "the art or practice of concealing a message, image, or file within another message, image, or file." The general premise is that your message is in plain sight, but obscured by the fact that no one, except who you want, knows to look for it. An image file or a block of text looks more unsuspecting than something like
03b062766c06092b6926f84ef0c41ad434fdfb327b6ee80c8fff87cefa09590f2212bb82b6b5

aa027a17529deadb99b206e580a3625f8784726d308bb9d7afa3e8cd97d83fb8f6ed1111c2ce

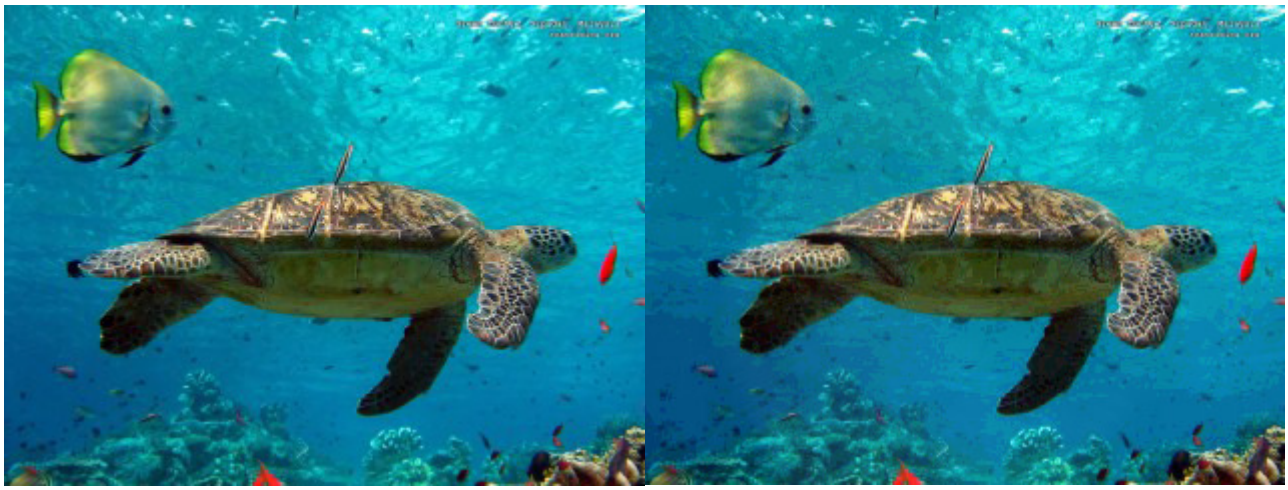
c4b64a60a5deca3bbaeba1b3241bb13718779ddaf01cd511f74c5ca59d1a51f11171cb9221cea9

ed6aad68fa73d22568899d328e

which is a 1024-bit RSA ciphertext. Steganography can be performed in countless ways -- from modifying bits in files to using invisible ink to write a message on a physical canvas. Steganography also has quite a history, going back to the times of ancient Greece. Herodotus wrote about Histiaeus shaving the head of his most trusted slave and tattooing a message on the slave's bald head. When the slave's hair regrew, the message would be concealed and the slave could be sent off without having to conceal any physical evidence of a message. The receiving end of the message (Aristagoras) would either know of the messages existence, or the slave would be trusted to reveal it to him upon arrival. With the advent of the digital age, Steganography can take advantage of technology and become more widespread. Images, text documents, music files, or any seemingly common and innocent looking object can be taken advantage of to conceal a message. This can then be transmitted over the internet to knowing parties without anyone knowing -- unless they are specifically looking for it. The sheer size of the internet makes steganographic messages very difficult (almost impossible?) to spot. Knowing that there are messages buried within the billions of images, or files in general, on the internet doesn't really provide any sort of start on how to get those images. Also, once a steganographic message is found, it is very difficult to extract. If the message was not encoded naively, there is not much of a starting point. The message can be encoded in a way to still pass a chi-square test for randomness of bits so extracting a key or plaintext message would be difficult. The steganographic message itself can be ciphertext which yields even further complications. As a result, an analogue to cryptanalysis called steganalysis has been (and is being) developed to find and extract steganographic codes. Most of the techniques rely on statistical analysis to find unusual features in a message -- which may go on to yield how it was encoded and allow for the original message to be deciphered.

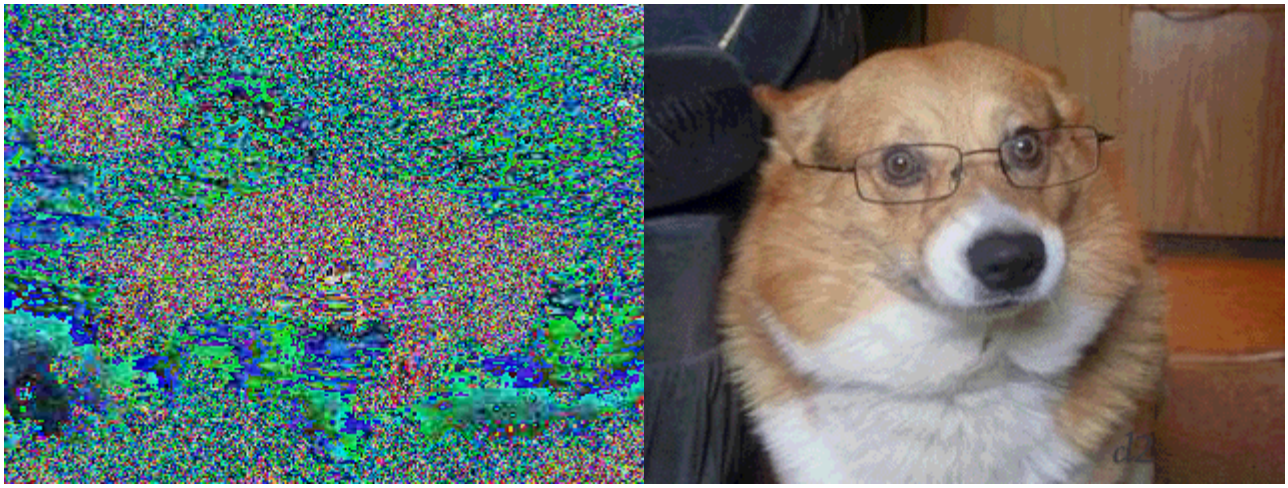
I recently decided to make a program that would hide one image in another. This works by doing a simple least significant bit substitution. The four least significant bits (LSBs) of every RGB value in the source image would be substituted with the four most significant bits (MSBs) of the hidden image. This takes advantage of the fact that the the four LSB values do not encode a large range of colors -- their range is 0 to 15, as opposed to 16 to 255 of the remaining four bits. The theory is then that you can take two pixels, line them up, and make a new pixel. This new pixel has the four MSBs of the original image, but has its four LSBs replaced by the four MSBs of the hidden image. This preserves most of the color range of the source image, and encodes the important part of the hidden image. Thus,

Visually, the difference is very small, even at the pixel level. On a large scale, with the colors blending together, a normal image and one that contains a message is very difficult to spot. The program that I originally wrote to do the simple LSB substitution technique produced these two images.



Which one has the hidden message embedded inside?

The difference in color is extremely difficult to spot on this scope. No image artifacts are present unless someone is specifically looking for them. At first glance, no one would know that a message is hidden in one of those two images. Performing the bit extraction on the following two images produces these two resulting images:



Quite a surprising difference. The results go back to the wide range that the four MSBs can encode.

A problem is that extraction is quite simple. An obfuscation technique that I decided to incorporate into this was the use of magic squares. A magic square is an n x n matrix containing integers whose sum across the rows, columns, and diagonals all equal the same value. For example, the following magic square sums up to 65:

$$\begin{vmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{vmatrix}$$

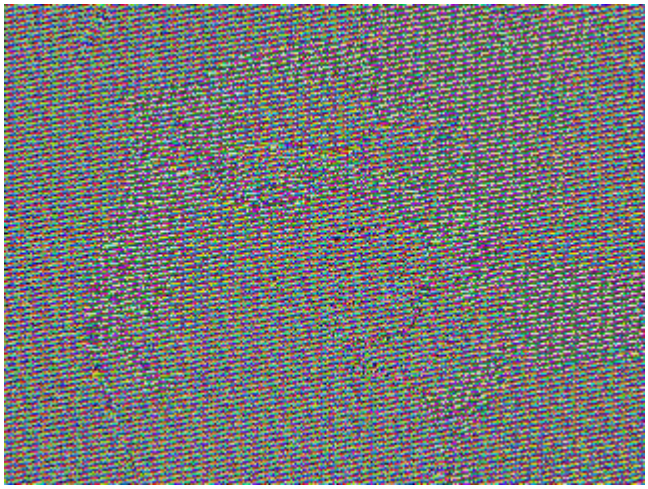
These squares are useful in that for odd numbered sizes there is a general formula to generate a number m at row i and column j. This provides for a linear time algorithm as a function of size and squares of any odd order can be quickly generated. Unfortunately, no formula exists for generating an even numbered square, so I avoided using them for this project. Sidenote: there are techniques to generate even squares in special forms. One of the interesting things about magic squares is the number of unique squares. Excluding rotations and reflections of a square, the number of unique square configurations is: 1, 0, 1, 880, 275305224, for n = 1 .. 5. For n >= 6 there is no known number of configurations. This provides an ample key space to choose from.

There are several ways that magic squares can be useful in steganography. Using a formula to generate a base magic square provides a unique encryption key. Alternatively, a magic square unique from the base square can be found and used as a key. This key can then be used to add a layer of encryption to the hidden data. For example, let the tile below be a 3x3 subsquare of a 9x9 square. A magic square key can then be applied by continually tiling squares over box pixel regions.

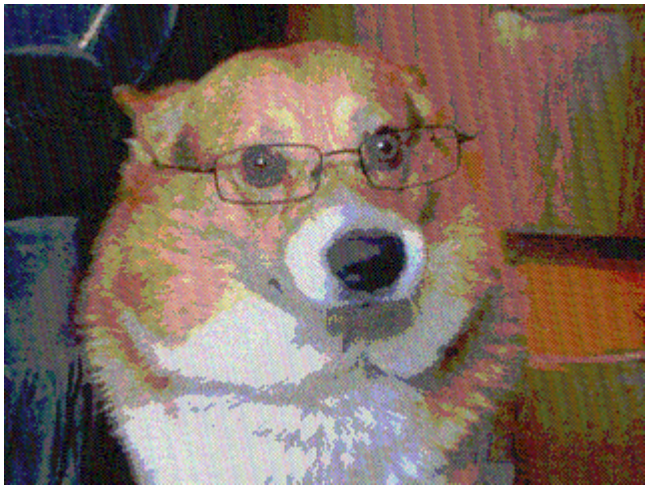
So the 9x9 square would be broken up into nine 3x3 squares and the exclusive-or operation applied to the MSB of the hidden image. The same square can then be used when extracting the hidden image. Alternatively, the square can be treated as a one dimensional array and each element will act as an exclusive-or value for each pixel in the hidden image. Taking it even further, the square can be continually rotated or reflected about axes so the key doesn't show as much repetition, although these steps would also have to be encoded in the image unless the same type of rotation/reflection takes place at each step. There are no shortage of possibilities for unique usage of magic squares. One question may be that since only four bits per color are stored, why use any sized square with values > 15 (0b1111). The answer is that these squares can

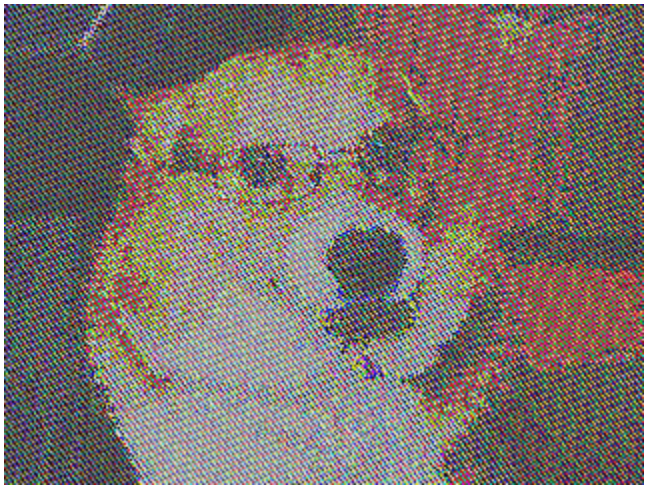
be extended to be used with any form of data -- not just images. Larger sized squares might be useful when more data can be stored -- perhaps at some block level instead of a byte level.

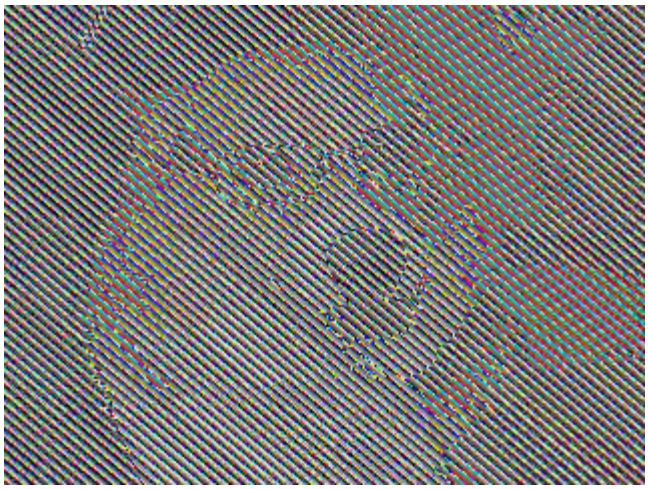
For simplicity, I chose to implement was a simple algorithm that treats a square as a one dimensional array and uses the four lowest bits of it as an XOR key against the MSBs of the hidden image. The result of extracting the four MSBs without knowing the key of the hidden image is shown below:



The result is significantly different without knowing the key. Extracting the four LSBs is not enough to give much information about the source image. The size of the square also affects how the resulting image will be viewed (which should be obvious). A longer key provides for more distortion across the image. A key of length is equal to or greater than the data to encrypt would provide the best results. Below are the result with magic squares of n = 7, 13, 17, and 51.







Overall, the technique is really simple to implement and there shouldn't be stopping anyone from using a simple XOR-based encryption for extra security, at the very least.

A copy of this post is available as a downloadable PDF here.