Hiding Data in Images

A Brief Introduction to Steganography

A Guide by Striker Security
https://strikersecurity.com

This guide contains a full four-part series of articles I wrote for the Black Hills Information Security blog about steganography and error correcting codes.

If you're interested in the original series, you can find it at: http://www.blackhillsinfosec.com/?p=5338.

As always, you can get in touch with me directly with an email to dakota@strikersecurity.com with any questions or comments. I always love hearing what you want to see next.

Happy hiding!

Dakota

Contents

Part 1: Image Formats

What if I told you this adorable puppy was hiding a secret message?

In this post, we'll find out how this dog was convinced to hide a message for us... and how to learn its secrets. Along the way, we'll learn a lot about how images work and just enough math to make your high school teacher say "I told you so."

Let's start with the basics. You might already know some of this, but stick with me here.

Computers are really just things that take bits:

```
01100010 01101100 01100001 01100011 01101011 01101000 01101001
01101100 01101100 01110011 01101001 01101110 01100110 01101111
01110011 01100101 01100011 00101110 01100011 01101111 01101101
00101111 01101100 01101100
```

And turn them into other bits:

```
01101010 01101111 01101000 01101110 00100000 01110010 01101111
01100011 01101011 01110011
```

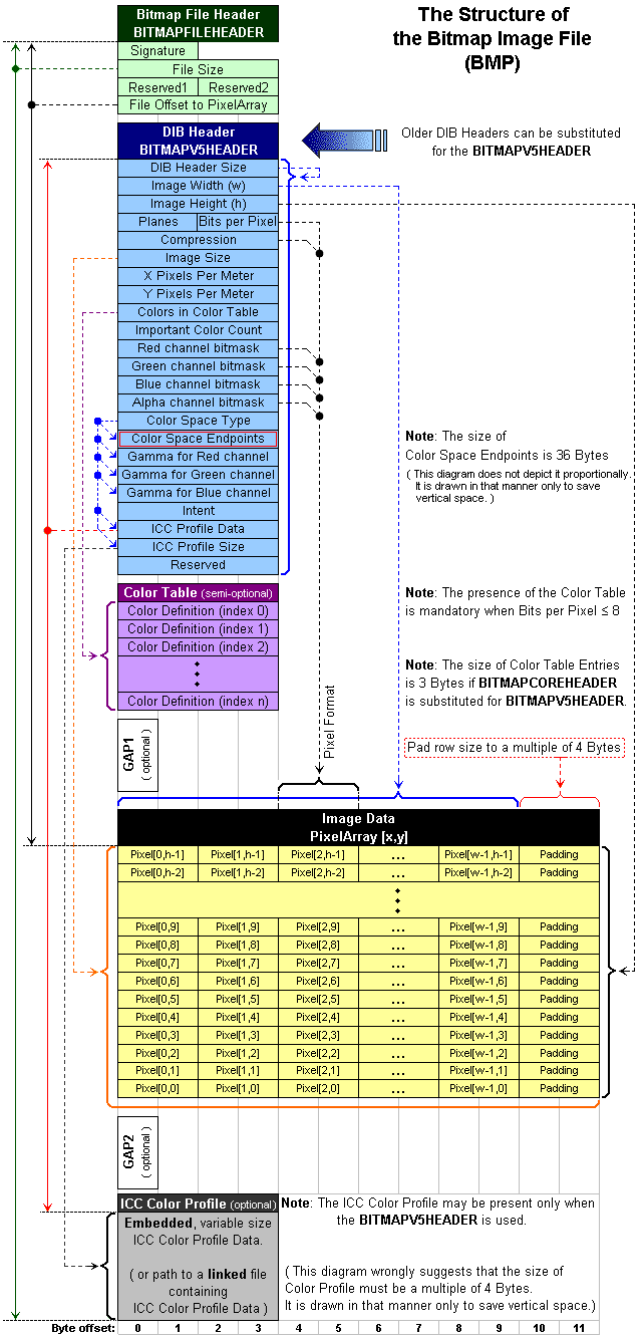
This makes up the core of information processing in the world. Underneath cell phones, computers, the Internet, everything digital - it's all just patterns of ones and zeros.

Unfortunately, though, not very many people can read binary, and so those bits have to be turned into something that actual humans can understand (since, for now at least, computers exist to serve humans).

This creates problems. Who says what patterns of bits turn into what words and pictures on a screen? Nobody, that's who. Er, also kind of everybody. Or maybe just some certain special people? Turns out, it's a huge mess. Various people and groups, at various times, for various reasons, have stuck a flag in the ground and said "THIS is how you turn a bunch of bits into an image of a cat!" Whenever you see a hilarious GIF, for example, you can thank the fine folks who worked at CompuServe in 1987 and decided how GIFs should work (but not how we're supposed to pronounce GIF, for some reason). Wanna know more than you ever wanted to about GIFs? Here's their full specification. It's alright, I'll wait.

Now that you've memorized the entire GIF format (you did, right? there will be a quiz later), we can move on. What does any of this have to do with hiding things? Well, we need a place to hide. Think of it as scoping out the best hide-and-seek locations in the new office.

For instance, in the bitmap format (what we'll be dealing with for the rest of this article; bitmap files end in `.bmp`), each pixel contains an R, G, and B (red, green, and blue) value, each of which are one byte (eight bits). Since you can combine red, green, and blue into any color if you mix them right, this means that just those three values can allow a pixel to be any color. When you get a big list of pixels, and decide how to shape the list (i.e. are those 500 pixels a 100 x 5 pixel image, or a 50 x 10 pixel image, or...) then you can display that long list of values as an image. Why will we use bitmaps here? Because they're incredibly simple.


The Structure of the Bitmap Image File (BMP)
Bitmap File Header
BITMAPFILEHEADER
Signature
File Size
Reserved1
Reserved2
File Offset to PixelArray
DIB Header
BITMAPV5HEADER
Older DIB Headers can be substituted for the BITMAPV5HEADER
DIB Header Size
Image Width (w)
Image Height (h)
Planes
Bits per Pixel
Compression
Image Size
X Pixels Per Meter
Y Pixels Per Meter
Colors in Color Table
Important Color Count
Red channel bitmask
Green channel bitmask
Blue channel bitmask
Alpha channel bitmask
Color Space Type
Color Space Endpoints
Gamma for Red channel
Gamma for Green channel
Gamma for Blue channel
Intent
ICC Profile Data
ICC Profile Size
Reserved
Note: The size of Color Space Endpoints is 36 Bytes
(This diagram does not depict it proportionally. It is drawn in that manner only to save vertical space.)
Color Table (semi-optional)
Color Definition (index 0)
Color Definition (index 1)
Color Definition (index 2)
Color Definition (index n)
Note: The presence of the Color Table is mandatory when Bits per Pixel ≤ 8
Note: The size of Color Table Entries is 3 Bytes if BITMAPCOREHEADER is substituted for BITMAPV5HEADER.
GAP1 (optional)
Pixel Format
Pad row size to a multiple of 4 Bytes
Image Data
PixelArray [x,y]
Pixel[0,h-1] Pixel[1,h-1] Pixel[2,h-1] ... Pixel[w-1,h-1] Padding
Pixel[0,h-2] Pixel[1,h-2] Pixel[2,h-2] ... Pixel[w-1,h-2] Padding
Pixel[0,9] Pixel[1,9] Pixel[2,9] ... Pixel[w-1,9] Padding
Pixel[0,8] Pixel[1,8] Pixel[2,8] ... Pixel[w-1,8] Padding
Pixel[0,7] Pixel[1,7] Pixel[2,7] ... Pixel[w-1,7] Padding
Pixel[0,6] Pixel[1,6] Pixel[2,6] ... Pixel[w-1,6] Padding
Pixel[0,5] Pixel[1,5] Pixel[2,5] ... Pixel[w-1,5] Padding
Pixel[0,4] Pixel[1,4] Pixel[2,4] ... Pixel[w-1,4] Padding
Pixel[0,3] Pixel[1,3] Pixel[2,3] ... Pixel[w-1,3] Padding
Pixel[0,2] Pixel[1,2] Pixel[2,2] ... Pixel[w-1,2] Padding
Pixel[0,1] Pixel[1,1] Pixel[2,1] ... Pixel[w-1,1] Padding
Pixel[0,0] Pixel[1,0] Pixel[2,0] ... Pixel[w-1,0] Padding
GAP2 (optional)
ICC Color Profile (optional)
Embedded, variable size ICC Color Profile Data.
(or path to a linked file containing ICC Color Profile Data)
Note: The ICC Color Profile may be present only when the BITMAPV5HEADER is used.
(This diagram wrongly suggests that the size of Color Profile must be a multiple of 4 Bytes. It is drawn in that manner only to save vertical space.)
Byte offset: 0 1 2 3 4 5 6 7 8 9 10 11

Yep, simple.

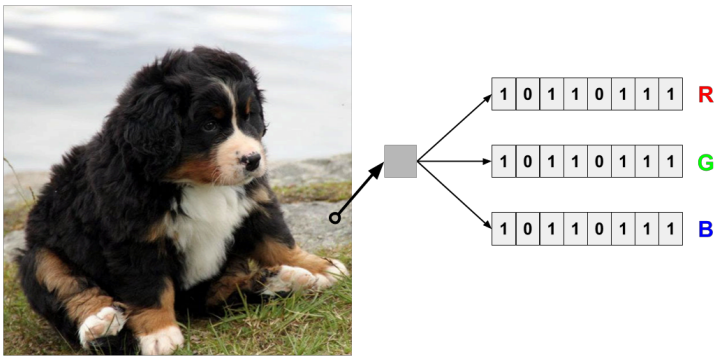
That means that this staggeringly lame 2x2 pixel image that I made just now:

Is actually this:

```
01000010 01001101 01000110 00000000 00000000 00000000  BMF...
00000000 00000000 00000000 00000000 00110110 00000000  ....6.
00000000 00000000 00101000 00000000 00000000 00000000  ..(...
00000010 00000000 00000000 00000000 00000010 00000000  ......
00000000 00000000 00000001 00000000 00011000 00000000  ......
00000000 00000000 00000000 00000000 00010000 00000000  ......
00000000 00000000 00000000 00000000 00000000 00000000  ......
00000000 00000000 00000000 00000000 00000000 00000000  ......
00000000 00000000 00000000 00000000 00000000 00000000  ......
11111111 00100110 00000000 00000000 00000000 00000000  .&....
00000000 00000000 00000000 00000000 11111111 00000000  ......
01111111 01011011 00000000 00000000                     .[..
```

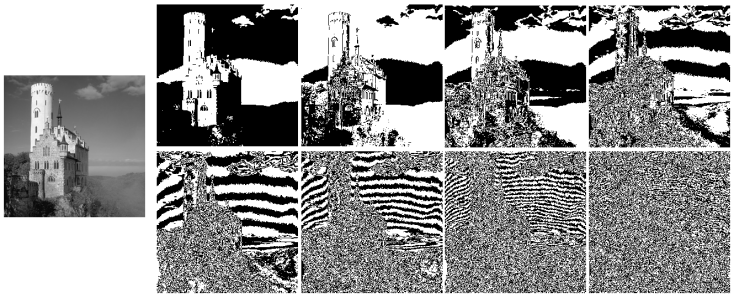
But why are we learning about this? We haven't even hidden anything yet! Fear not - we shall soon. First, we need to ruin some perfectly good pictures in the process of finding ourselves a place to hide.

One of the implications of using numbers for things is that all bits are equal, but some bits are more equal than others. If you have 5005, and you change the 5 on the left to a 6, that's a much bigger difference (a difference of a thousand) than if you have 5005 and change the 5 on the right to a 6 (a difference of one). The five on the left in the thousands column would be called the "most significant digit" while the five on the right in the ones column would be the "least significant digit," and picking which five to change matters a **lot**.



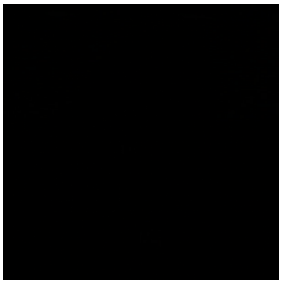
Let's do some quick review: images (at least the ones we'll be working with here) are made up of pixels arranged in a grid. Each pixel is made up of three values; R, G, and B. By mixing the red, green, and blue values, the pixel can be any color. Just like in "regular" numbers, the binary digits making up the pixels have different significance (matter more) the further to the left they are.

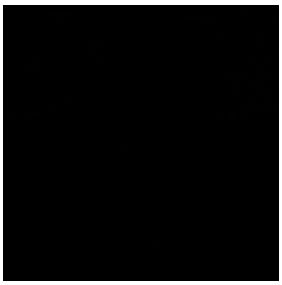
In images, the least significant bit in R, G, and B for each pixel does nearly nothing, while the most significant bit can really ruin your day. For instance, this is what happens when you take each of the 8 bits out of a black and white bitmap one layer at a time and make an image out of each layer. Each image represents one "significance level" of bits; the most significant bit is on the top left, and the least significant on the bottom right.



See how the most significant bit (top left) makes up most of the image, while the least significant bit (bottom right) is basically just random noise? I bet you could change all of those least significant bits (or maybe even the last two) and nothing would look different in the final image... perhaps you could change them in some sort of pattern... like in a message, say. Just a thought.

Here's what happens when we take that puppy and flip the least significant bits of every pixel (each of R, G, and B) to all be 1, then the last two bits to both be one, then the last three, and so on:





Did you notice how the first few look totally fine? So it's concluded: we can definitely flip the first couple of bits in each pixel value of an image, and change them however we want, and nobody will be able to tell.

We just found ourselves a place to hide.

Part 2: Hiding Data in Images

In part 1, we talked about how bits make up images, and what that means for our game of digital hide-and-seek. In this post, we'll take our new hiding place and put it to work hiding things, as one does. Now that we know where to hide, how do we actually take advantage of that knowledge? With programming, of course! The first thing we need is something to hide. I'll leave the more questionable part of that to you, and just use this snippet of Python instead, which will take some text and turn it into a list of bits:

```
# let's get our message set up
message = list('this is a message')

# convert to binary representation
message = ['{:07b}'.format(ord(x)) for x in message]
print("Message as binary:")
print(message)

# split the binary into bits
message = [[bit for bit in x] for x in message]

# flatten it and convert to integers
message = [int(bit) for sublist in message for bit in sublist]
print("Message as list of bits:")
print(message)
```

The final output of this should be a message that looks like this:

```
[1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0,
 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0,
 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0,
 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0,
 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1,
 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1]
```

Which is the phrase "this is a message" in binary. Woo! We have something to hide!

Now we have to take this image:

And put our message into it, hiding it in the least significant bits of the image.

We'll use this code snippet, which opens up an existing image and adds a message into it, repeating each bit in the message nine times for reasons that will become clear in a moment:

```
from PIL import Image, ImageFilter
import numpy as np

# first, open the original image
imgpath = 'images/original/image.bmp'
img = Image.open(imgpath)

# we'll use simple repetition as a very rudimentary error
# correcting code to try to maintain integrity
# each bit of the message will be repeated 9 times - the three
# least significant bits of the R,G, and B values of one pixel
imgArray = list(np.asarray(img))

def set_bit(val, bitNo, bit):
    """ given a value, which bit in the value to set, and the
    actual bit (0 or 1) to set, return the new value with the
    proper bit flipped """
    mask = 1 << bitNo
    val &= ~mask
    if bit:
        val |= mask
    return val

msgIndex = 0
newImg = []
# this part of the code sets the least significant 3 bits of the R, G, and B values
# in each pixel to be one bit from our message
# this means that each bit from our message is repeated 9 times - 3 each
# in R, G, and B. This is a waste, technically speaking, but it's needed in case
# we lose some data in transit
# using the last 3 bits instead of the last 2 means the image looks a little worse
# but we can store more data in it - a tradeoff
# the more significant the bits get, as well, the less likely they are to be
# changed by compression - we could theoretically hide data in the most significant
# bits of the message, and they would probably never be changed by compression or etc.,
# but it would look terrible, which defeats the whole purpose
for row in imgArray:
    newRow = []
    for pixel in row:
        newPixel = []
        for val in pixel:
            # iterate through RGB values, one at a time
            if msgIndex >= len(message):
                # if we've run out of message to put in the image, just add zeros
                setTo = 0
            else:
```

```
                # get another bit from the message
                setTo = message[msgIndex]
            # set the last 3 bits of this R, G, or B pixel to be whatever we decided
            val = set_bit(val, 0, setTo)
            val = set_bit(val, 1, setTo)
            val = set_bit(val, 2, setTo)

            # continue to build up our new image (now with 100% more hidden message!)
            newPixel.append(val) # this adds an R, G, or B value to the pixel
        # start looking at the next bit in the message
        msgIndex += 1
        newRow.append(newPixel) # this adds a pixel to the row
    newImg.append(newRow) # this adds a row to our image array

arr = np.array(newImg, np.uint8) # convert our new image to a numpy array
im = Image.fromarray(arr)
im.save("image_steg.bmp")
```

You're probably wondering... why are we repeating the message so much? Nine times per bit seems excessive.

It turns out that we aren't the only people who have noticed that the least significant bits in an image are basically random. Someone has beaten us to our own hiding place, and they're using it for boring stuff.

The objective of compression, according to Wikipedia, is "to reduce irrelevance and redundancy of the image data in order to be able to store or transmit data in an efficient form."

But that "irrelevant and redundant data" is where we wanted to put our sneaky message stuff, and compression destroys those bits. Drat. Turns out if there are useless bits, such as the least significant bit of each pixel value, they're perfect for hiding things in because nobody cares about them, but also the first to get thrown out by compression... because nobody cares about them.

So we fight back, by repeating ourselves a bunch so that even if some bits get flipped by compression, our data still mostly makes it through. It's not elegant, but it works. (This will be better explained in part 3, where we'll get into more elegant methods using some cool math.)

Once we run the image through our code, it looks like this:

Which might look familiar - and now we know the message that this puppy is hiding from part 1! But... how do we get it out once it's been put in?

Here's how:

```
# open the image and extract our least significant bits to see if
# the message made it through

img = Image.open(path)
imgArray = list(np.asarray(img))

# note that message must still be set from the code block above
# (or you can recreate it here)
origMessage = message[:20] # take the first 20 characters of the original message
# we don't use the entire message here since we just want to ensure it made it through
print("Original message:")
print(origMessage)

message = []

for row in imgArray:
```

```
    for pixel in row:
        # we'll take a count of how many "0" or "1" values we see and then go with
        # the highest-voted result (hopefully we have enough repetition!)
        count = {"0": 0, "1": 0}
        for val in pixel:
            # iterate through RGB values of the pixel, one at a time
            # convert the R, G, or B value to a byte string
            byte = '{:08b}'.format(val)
            # then, for each of the least significant 3 bits in each value...
            for i in [-1, -2, -3]:
                # try to get an actual 1 or 0 integer from it
                try:
                    bit = int(byte[i])
                except:
                    # if, somehow, the last part of the byte isn't an integer...?
                    # (this should never happen)
                    print(bin(val))
                    raise

                # count up the bits we've seen
                if bit == 0:
                    count["0"] += 1
                elif bit == 1:
                    count["1"] += 1
                else:
                    print("WAT")

        # and once we've seen them all, decide which we should go with
        # hopefully if compression (or anything) flipped some of these bits,
        # it will flip few enough that the majority are still accurate
        if count["1"] > count["0"]:
            message.append(1)
        else:
            message.append(0)

# even though we extracted the full message, we still only display the first 20
# characters just to make sure they match what we expect
print("Extracted message:")
print(message[:20])
```

Run this on the image, and you get the first 20 characters of the original message and newly-extracted message:

```
Original message:
[1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0]
Extracted message:
[1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0]
```

Awesome! They're the same! We just moved data around hidden in an image using steganography! (Something to try on your own: can you reassemble these bits back into text by reversing the process from earlier?)

Being able to extract steganographically encoded data from an image is cool, but having to repeat ourselves so much means that we can't move very much data, and that it's fairly obvious - the image with hidden data in it looks different enough from the original that you can tell something is up if you look closely enough. This image is 500 by 500 pixels, which means (since we can only hide one bit of data per pixel) that we can only hide just over 31 kB of data in this image. That's great, and somewhat useful, but you're going to need a lot of pictures to send any significant amounts of data - especially since we're using the least significant 3 bits in the image, and we'd prefer to use less so that the image doesn't look any different. In part 3, we'll explore how to use more complicated error correcting codes to make our data hiding more efficient.

Part 3: Error Correcting Codes

This is part three of a four part series. In part 1, we covered the basics of image formats and found a place to hide data in images. In part 2, we actually wrote code to steganographically encode data into an image and then extract it without making the image look different.

Tired of this picture yet?

We ran into a problem in part 2, though - compression. When images are compressed, our carefully hidden data can be damaged. We fixed that by repeating every bit in the message 9 times, but we can do better. This post will cover how we can use some math, in the form of error correcting codes, to hide more bits of data in each image.

Our friend Wikipedia gives us a good definition of what an error correcting code is:

"An error-correcting code (ECC) is a process of adding redundant data to a message, such that it can be recovered by a receiver even when a number of errors (up to the capability of the code being used) were introduced."

The repetition we used earlier is the simplest possible ECC - we send redundant

data, repeating ourselves over and over, so that even if some of the bits are wrong we can still receive the message.

Before we start figuring out something better, though, we need to define our problem. We know that compression destroys data, but how?

Firstly, we can say that compression introduces *noise* - in other words, some bits are not what we expect them to be. For instance, we might put the message `1101` in and get `1100` out - in which case, we would say that the last bit was "flipped" from a 1 to a 0. Now that we know we have some noise, how can we define it?

Well, we have several options here. You can read papers, like this 8 page one written by someone at the Fleet Information Warfare Center, or you can make some assumptions and see if they work.

Let's compromise and make some assumptions that are informed by reality. They might not be 100% right, but they're right enough. So, for our sake, we can assume:

1. Random, position-independent noise: there are no specific patterns in the noise; any given bit has the same chance of being flipped as any other bit of the same significance. This means it doesn't matter *where* in the image we encode data; a pixel in the bottom left hand corner of the image (for example) is as likely to be changed as a pixel in the middle.

2. More significant bits have less noise: the least significant bit of any pixel is the most likely to be flipped since compression tries to eliminate useless data, and the chance that a bit will be flipped decreases as the bits become more significant.

3. The noise is binary symmetric: this is a fancy way of saying that the noise is unbiased and evenly distributed - a zero is just as likely to be flipped as a one.

Rather than spend a lot of time uploading and downloading images from web sites where they'll be compressed, we can simulate this type of noise by randomizing some bits in the image. This lets us use code, which is nice and controllable so we can do lots of testing easily.

Here's our image before and after blurring:

It may not look very different, but it turned `we're going to hide this message in an image!` into `we're eking to hide this message in an image` (without the exclamation point on the end), so it's definitely causing some damage.

The challenge now is to be able to extract data from the blurred image, even though there's noise. The repetition from part 2 works, but having to repeat every bit 9 times means we can't transfer nearly enough data. We can do better.

Before we can dive into error correction, though, we need to talk about parity bits. Essentially, a parity bit adds a check to any given binary string that tells us whether the number of 1-bits in the string is odd. For instance: the string `000` has no 1-bits, so its parity bit would be 0 (leaving us with the final string `0000`). In contrast, the string `010` has an odd number of 1-bits, so its parity bit is 1, leaving us with `0101` . You'll notice that this means every string has an even number of ones in it once the parity bit is added.

The process is:

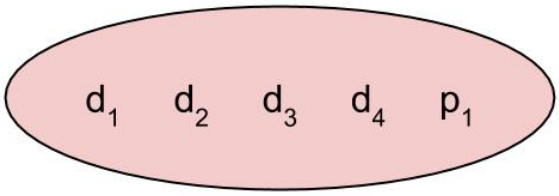
1. Compute parity bits based on your message. If it has an odd number of `1` values, the parity bit is `1` , otherwise the parity bit is `0` . 2. Put those parity bits into the message; they're frequently on the end, but it doesn't actually matter where they are so long as the receiver knows where to find them.

Why do we do this? Simple - if a bit gets flipped somewhere in the string, we can tell something is wrong by checking the parity bit (if the string `0101` from above comes across as `1101` , we know that the parity bit of `110` should be 0 - but it's 1, so something must be wrong). Unfortunately, this only tells us that something is wrong - not what. We have no way of knowing which bit was flipped.

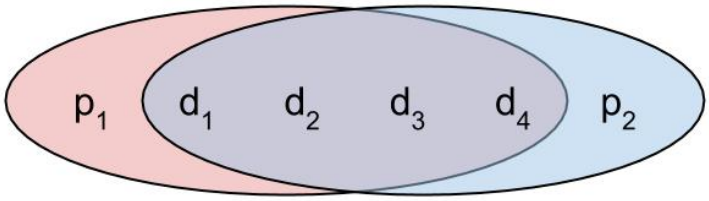
Enter Richard Hamming, who invented Hamming codes, a way of combining parity bits to both *detect* and *correct* errors. Hold on, this is about to get awesome (and really math-y).

For this next part, let's assume you're trying to send 4 bits of data, and you know (somehow) that you won't have more than one error in them.

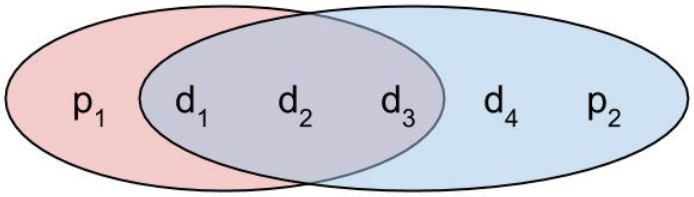
Say you've got your usual parity bit with the 4 bits of data. If there's an error, how can you tell which bit is wrong? You can't - one of the four is flipped, but there's no way to know which. (Note that the circles in these graphics each represent a parity group - once all the parity bits are computed and added, each will have an even number of ones in it. This means that if a circle *doesn't* have an even number of ones in it when we receive the message, we know something is wrong.)



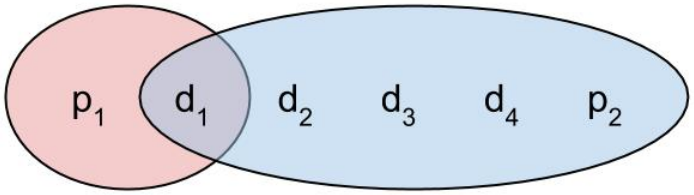
If you just add another parity bit, that's great, but all you've done is repeated the parity bit - which is useful, since it means you can maybe tell if either of them is wrong by checking the other. But this mostly seems like a waste. You still can't tell which data bits have been flipped if anything goes wrong - the second parity bit provides no new information.



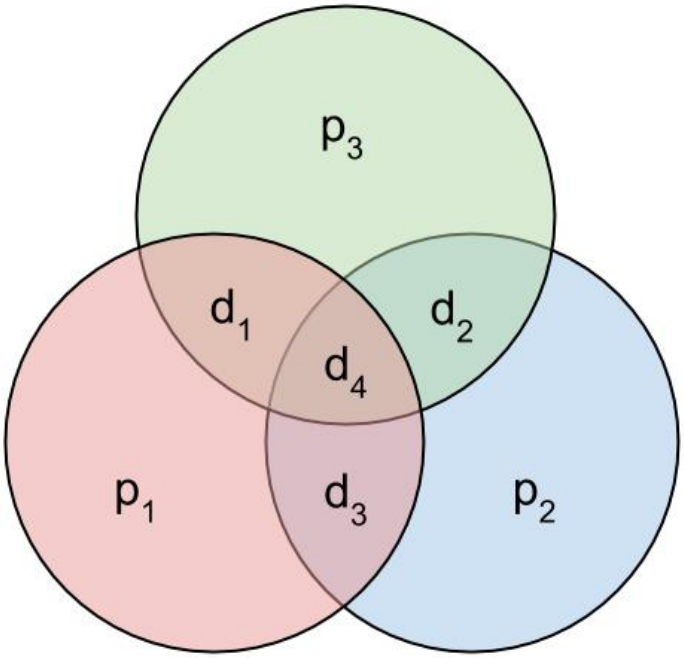
Let's try something weird. What if you exclude the 4th data bit from the first parity bit? If either of the first three data bits are flipped, the first parity bit will tell us, and if any of them are flipped, the second parity bit will tell us as usual, but... if the first parity bit is wrong, it can be any of the first three data bits that are wrong, but if the second parity bit is wrong, it can be any of the four. Seems like we're kind of narrowing in on something here.



If we take this one step further and keep shrinking the number of bits covered by the first parity bit, we get to the layout below. We've done an incredible thing here! If both parity bits are wrong, we know that the first data bit is wrong - not just "a bit," but *exactly which bit is wrong!*



If you take this further, you can end up here:



Seriously, take a minute to look at the image above. This means that if *any one of the data bits is flipped, you can tell which it is and fix it*, because you know that each circle must contain an even number of ones. By extending this idea, you can play with the number of parity bits and data bits to handle different levels of error - broadly speaking, more parity bits help you correct more errors, but it means you have less space for data.

If error correcting codes are still confusing, think of it this way - we've created a bunch of potential messages which are *invalid to receive*, so that if you get them, you know something is wrong (the valid messages are known as "code words" - the "code" in "error correcting codes"). When you get one of these messages that you know is wrong, you can correct for errors by choosing the valid code word that is most similar to what you received (which means we want code words to be as different as possible so that it's obvious which code word any given incorrect message is supposed to be).

This is the math behind correcting errors in a much more efficient way than the simple repetition we've used before - in our case, we can fit more than four times the amount of data! Next time, we'll take this math and put it to work.

Part 4: Resilient Steganography

This is it. The end. The last of a four part series covering image steganography. You can get started with part 1, part 2, or part 3, or dive right in below:

We've covered the basics of hiding data in images. We've covered the math. We've covered error detection and correction using hamming codes. There have been Venn diagrams, and puppies, and secret messages, and it's all been very exciting.

Where do we end?

With code, of course! Let's tie everything together, slap a nice bow on it, write some documentation, and declare our steganography project complete. I left you after part 3 with a bunch of math about error correcting codes, and a promise to put that math to work, and that's exactly what we'll do - steganographically hide data which we've run through an hamming encoder so that we can correct errors caused by compression (or anything else).

All of the code for this article can be found on Github - I'll walk you through it below, but you can always go take a look for yourself.

Let's get started.

When you read part 2, we covered the Python code used to steganographically insert data into images. The final steganography software, in Python, which does this is the same as before (except slightly cleaned up), so I'll skip over a full explanation of it. As a refresher - the least significant bits of each pixel in the image are basically random, so we can change them to whatever we want and the image will look the same to a human observer. This code just flips the correct bits in order to output an image with our message in it.

What happens, though, if the image is then compressed, or damaged in some way? In the last post, we walked through the math behind hamming codes, an error-correcting code that allows us to fix errors in our message. Here's the code that puts that math into action:

```
def encode(msg):
    """ passed a list of bits (integers, 1 or 0), returns a
    hamming(8,4)-coded list of bits """
    while len(msg) % 4 != 0:
        # pad the message to length
        msg.append(0)

    msg = np.reshape(np.array(msg), (-1, 4))

    # create parity bits using transition matrix
    transition = np.mat('1,0,0,0,0,1,1,1;\
                         0,1,0,0,1,0,1,1;\
                         0,0,1,0,1,1,0,1;\
                         0,0,0,1,1,1,1,0')

    result =  np.dot(msg, transition)

    # mod 2 the matrix multiplication
    return np.mod(result, 2)
```

This is some pretty dense code, so let's walk through it one piece at a time. First, we add zeros to the end of the message until it's the proper length so that the matrix multiplication will work out right (the number of bits in the message must be a multiple of 4).

Once the message is the right length, we create an nx4 array of the message's bits (where n is whatever it has to be to fit the whole message). This array is then multiplied by a transition matrix.

Hold up.

"Matrices," you say, "where did they come from? We haven't talked about any stinkin' matrices."

Well, astute reader, you caught me. I didn't mention the matrices, and I'm going to mostly ignore them here, except to say this: remember when we had to count up the number of bits in each circle of the Venn diagram and then create a parity bit for each group based on what the data bits were? That's what this matrix does. We multiply the message by this hamming code matrix, then mod the result by two (that is, take the remainder of each entry in the resulting array divided by 2) and we have ourselves a hamming-encoded message.

(If you still don't like me not explaining the matrix multiplication, here's the mathy version: the left half of the matrix is the identity matrix (preserving our

original message), while the right half's columns are entirely linearly independent from each other such that every generated parity bit is based on 3 data bits with no redundancy in parity. Given 4 bits of data, this matrix outputs 8 bits of "data plus parity," known to error correcting code people as a codeword.)

Now that we've got a hamming-encoded message that will tolerate some errors, we insert it into an image, as usual, exactly how we've discussed in previous posts. We then extract it on the other end - again, as usual. The mechanics of actual steganography should be pretty familiar to you by now. (If not, go back and read part 2 for a discussion of image steganography techniques.)

Once we've retrieved our message from the image that our steganography algorithm put it into, how do we fix errors? That's the whole point of this hamming error correction code thing, after all.

It turns out that the answer is more matrices. This next piece of code acts as a hamming code decoder in three parts. We'll break each down individually.

```
def syndrome(msg):
    """ passed a list of hamming(8,4)-encoded bits (integers, 1 or 0),
        returns an error syndrome for that list """

    msg = np.reshape(np.array(msg), (-1, 8)).T

    # syndrome generation matrix
    transition = np.mat('0,1,1,1,1,0,0,0;\
                         1,0,1,1,0,1,0,0;\
                         1,1,0,1,0,0,1,0;\
                         1,1,1,0,0,0,0,1')

    result = np.dot(transition, msg)

    # mod 2 the matrix multiplication
    return np.mod(result, 2)
```

The first task is to calculate a syndrome for this hamming code. This error syndrome, as it's called, is basically a record of what's wrong with the message - much like the syndrome of a disease, it can tell us what's wrong with the message so we can tell how to apply our error corrections to fix it. The mechanics here are the same as the hamming encoding - get the array in the right shape, multiply with the proper hamming code syndrome matrix, then mod everything by 2.

```
def correct(msg, syndrome):
    """ passed a syndrome and a message (as received, presumably with some
        errors), will use the syndrome to correct the message as best possible
    """

    # the syndrome for any incorrect bit will match the column of the syndrome
    # generation matrix that corresponds to the incorrect bit; a syndrome of
    # (1, 1, 0, 1) would indicate that the third bit has been flipped, since it
    # corresponds to the third column of the matrix

    # syndrome generation matrix (copy/pasted from above)
    transition = np.mat('0,1,1,1,1,0,0,0;\
                         1,0,1,1,0,1,0,0;\
                         1,1,0,1,0,0,1,0;\
                         1,1,1,0,0,0,0,1')

    for synd in range(syndrome.shape[1]):
        if not np.any(syndrome[:,synd]):
            # all zeros - no error!
            continue

        # otherwise we have an error syndrome
        for col in range(transition.shape[1]):
            # not very pythonic iteration, but we need the index
            if np.array_equal(transition[:,col], syndrome[:,synd]):
                current_val = msg[synd,col]
                new_val = (current_val + 1) % 2
                msg.itemset((synd,col), new_val)

    return msg
```

Once we have a syndrome, we know how to correct the message. This code above matches each syndrome to the bit that needs to be flipped by comparing each error syndrome to the syndrome generation matrix from above. If we find a match, we flip the corresponding bit - if it doesn't match, we use the continue keyword to skip to the next iteration of the loop.

```
def decode(msg):
    r = np.mat('1,0,0,0,0,0,0,0;\
                0,1,0,0,0,0,0,0;\
                0,0,1,0,0,0,0,0;\
                0,0,0,1,0,0,0,0')

    res = np.dot(r, msg.T)

    # convert to a regular python list, which is a pain
    return res.T.reshape((1,-1)).tolist()[0]
```

Now that we've corrected our message, we can decode it! Using the decoding matrix above, we do the same ol' matrix multiplication in order to get our final message out.

So, what's the outcome of all this? Continuing our image steganography example from part 3, this is the final result:

Here's our original message, with text on the right and the corresponding hex values on the left:

```
[dnelson@blueharvest hamming-stego]$ xxd -s 7 -l 45 output.txt
00000007: 7765 2772 6520 676f 696e 6720 746f 2068  we're going to h
00000017: 6964 6520 7468 6973 206d 6573 7361 6765  ide this message
00000027: 2069 6e20 616e 2069 6d61 6765 21          in an image!
```

This is the data after some errors have been inserted into it and it's been padding a little bit:

```
[dnelson@blueharvest hamming-stego]$ xxd -s 70 -l 100 output.txt
00000046: 7765 2772 6520 656b 696e 6720 746f 2068  we're eking to h
00000056: 6964 6520 7468 6973 206d 6573 7361 6765  ide this message
00000066: 2069 6e20 616e 2069 6d61 6765 0100 0400   in an image....
00000076: 0000 0000 0000 0000 0002 0000 0000 0000  ................
00000086: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000096: 0000 0800 0000 0000 0000 1000 0000 0000  ................
000000a6: 0000 0000                                ....
```

And this is our final, corrected output:

```
[dnelson@blueharvest hamming-stego]$ xxd -s 54765 -l 100 output.txt
0000d5ed: 7765 2772 6520 676f 696e 6720 746f 2068  we're going to h
0000d5fd: 6964 6520 7468 6973 206d 6573 7361 6765  ide this message
0000d60d: 2069 6e20 616e 2069 6d61 6765 2100 0000   in an image!...
0000d61d: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d62d: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d63d: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d64d: 0000 0000                                ....
```

That's it - we corrected a message using a hamming code! Whereas before we had to repeat each character 9 times, this hamming code fits 4 bytes of data into each 8 byte code word - a 1:2 ratio of data to total instead of our 1:9 from before. A pretty sizable improvement!

Note, however, that the error correction capabilities of a hamming code are only so good. The "damaged" message up above is still pretty readable to a human - much more than that, and errors start to sneak through to the end. Maybe that's okay... but maybe it's not. As always, there are tradeoffs.

Want to see the full image steganography example in action? Visit https://github.com/DakotaNelson/hamming-stego and check it out! That link heads straight to Github, where you'll find a free steganography tool in Python, usable on Linux, Mac, Windows, or anywhere else you can run Python.