

Java Art Chapter 6. Steganography

Steganography is a familiar idea to readers of detective and spy novels. It involves the hiding of a secret message inside an innocent-looking package or container (often called a carrier). For example, a micro-dot hidden beneath a postage stamp, or a message written in milk on the back of a letter, or instructions tattooed under a person's hair.

Fortunately, this chapter isn't about an uncomfortable visit to a local tattoo parlor to have "I love Java" emblazoned on my scalp. Less sensationally, it's about using steganographic techniques to hide text inside PNG images.

Figure 1 shows the idea – the two images appear to be the same, even when compared minutely. However, the right-hand image has the text of a Java program hidden inside it.



Figure 1. Two Images – Before and After Hiding Program Text.

Steganography is sometimes confused with *cryptology*, not surprisingly since it's a closely related idea. Cryptology scrambles a message, so an uninvited reader is unable to understand it. Steganography is about secrecy, so a potential eavesdropper won't even know there's a message to be read.

Steganography and cryptology can be combined to produce a hidden encrypted message – two levels of protection from prying eyes, which I'll implement using the Jasypt API (<http://www.jasypt.org/>). Jasypt supports basic encryption, without the programmer requiring a Masters degree in cryptology.

Another topic related to steganography is *digital watermarking*, which is employed for tracing and identifying digital media, such as images, audio, and video.

The cracking of steganographic messages is called *steganalysis*, and comes in two main forms. The easiest type of cracking simply makes the hidden message unreadable by modifying the carrier. This can usually be achieved by cropping or blurring the image, or saving it in a different file format. A much harder task is the extraction of the hidden message, which typically starts with the identification of tell-tale regularities or patterns in the carrier, or spotting differences between the carrier and its original. I'll discuss some basic steganalysis techniques using ImageJ (<http://rsbweb.nih.gov/ij/>), Java-based image processing software.

Protecting the hidden message when the carrier is modified is a hard problem. I'll implement two strategies which help a little: the duplication of the message in several

different parts of the image, and the splitting up (fragmentation) of the message into multiple pieces. These techniques help to hide the message better, and can withstand (to some extent) the carrier image being cropped.

A good place for more information on steganography is its Wikipedia page (<http://en.wikipedia.org/wiki/Steganography>), which includes links to steganographic software. Neil Johnson's *Steganography and Digital Watermarking* webpage at <http://www.jjtc.com/Steganography/> is another great starting point, and links to his list of over 100 steganography tools.

How does steganography fit into my "Java Art" theme for these chapters? Back in chapters 1 and 2 my aim was to convert a program into an image, resulting in a rather pixilated, abstract picture. With steganography, a program can be hidden inside any image (so long as it's big enough).

This chapter describes four steganography applications:

- `Steganography.java`: contains a basic set of steganography methods for hiding text inside a PNG image, and for extracting the text later.
- `StegCrypt.java`: a modification of the `Steganography` class to support encryption and decryption. It utilizes the `Jasypt` API.
- `MultiStegCrypt.java`: it stores multiple copies of the encrypted steganography message (stego for short) inside the image.
- `FragMultiStegCrypt.java`: it splits the message into parts before storing the encrypted fragments multiple times inside the image. This variation of `MultiStegCrypt` improves the stego's resistance to image cropping.

1. LSB Steganography

The simplest form of digital steganography (and probably the most common) is the Least Significant Bit (LSB) method, where the binary representation of the data that's to be hidden is written into the LSB of the bytes of the carrier. The overall change to the image is so minor that it can't be seen by the human eye.

Figure 1 shows the first stage of the process, when the image data is accessed as a series of bytes. Depending on the image format, a pixel may be represented by one or more bytes. In my examples, I'll be using 24-bit PNG images, which use a byte each for the red (R), green (G), and blue (B) channels.

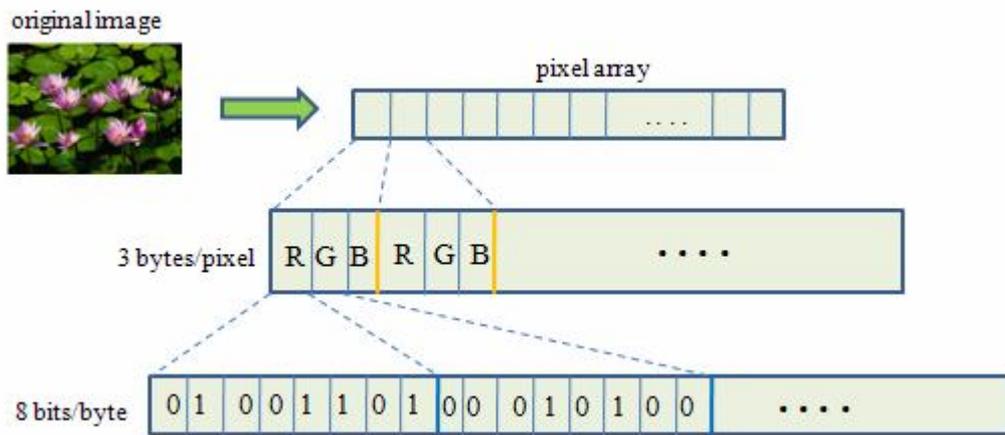


Figure 1. Accessing the Bits of a PNG image.

The next stage is to read in the text file, and access its bits, as shown in Figure 2.

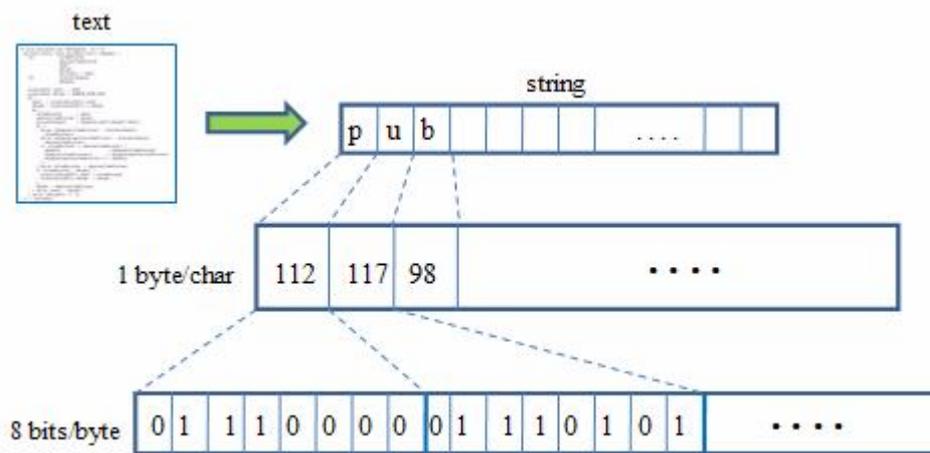


Figure 2. Accessing the Bits of a Text File.

Now its time to insert the bits of the text file into the image. The LSB approach only modifies the least significant bit of each image byte, as illustrated by Figure 3.

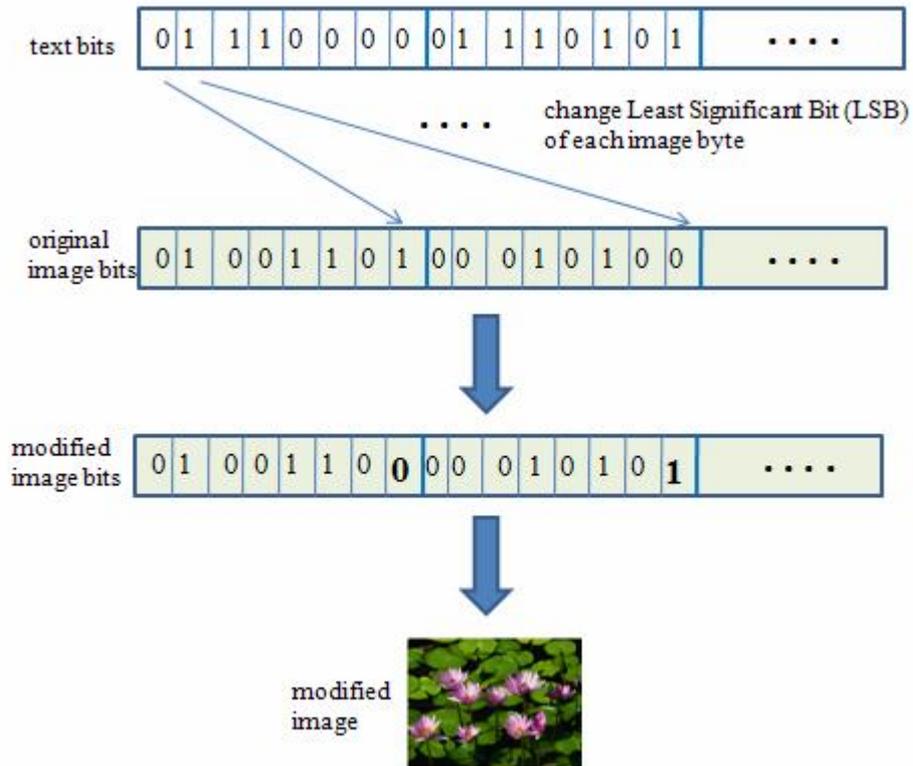


Figure 3. Inserting the Text Bits into the Image.

Extracting the text from the image at a later time involves copying the LSBs of the modified image's bytes, and recombining them into bytes in a text file, as in Figure 4.

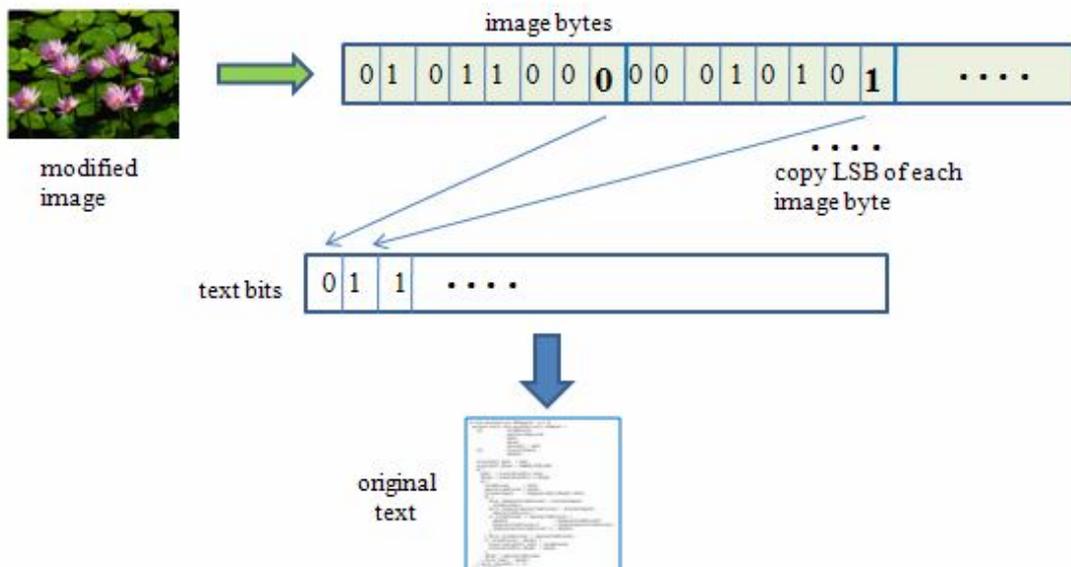


Figure 4. Extracting the Text from the Modified Image.

The LSB approach is the basis of all of my steganography classes.

2. Implementing LSB Steganography

The Steganography class implements the LSB method for hiding text inside a PNG image as explained in the diagrams above, but with one addition. The length of the text in binary form is calculated beforehand, and hidden in the image before the text. In other words, the steganographic information (the *stego*) has two parts: the size of the binary message, followed by the message itself.

The stego is spread over the image by modifying each byte's LSB. This means that 1 byte of stego data requires the modification of 8 bytes of the image (i.e. 1 stego data bit is stored in 1 image byte).

The size information is utilized when the text is extracted from the image, so the extraction process knows when to stop. The size data is a Java integer, which employs four bytes, and so needs 32 (4*8) bytes at the start of the image.

The Steganography class has two public static methods:

- `boolean hide(String textFnm, String imFnm);`
- `boolean reveal(String imFnm);`

2.1. Hiding a Message

`hide()` loads the text from the text file `textFnm`, the PNG image from the `imFnm` file, and stores the modified image in `<imFnm>Msg.png`.

```
public static boolean hide(String textFnm, String imFnm)
{
    // read in the message
    String inputText = readTextFile(textFnm);
    if ((inputText == null) || (inputText.length() == 0))
        return false;

    byte[] stego = buildStego(inputText);

    // access the image's data as a byte array
    BufferedImage im = loadImage(imFnm);
    if (im == null)
        return false;
    byte imBytes[] = accessBytes(im);

    if (!singleHide(imBytes, stego)) // im is modified with the stego
        return false;

    // store the modified image in <fnm>Msg.png
    String fnm = getFileName(imFnm);
    return writeImageToFile( fnm + "Msg.png", im);
} // end of hide()
```

`readTextFile()` reads in the text file, returning it as a string. `loadImage()` uses `ImageIO.read()` to load the image file, returning it as a `BufferedImage`.

`buildStego()` constructs a byte array consisting of the two field: the size of the text message in binary form, and the binary message itself.

```

// global
private static final int DATA_SIZE = 8;
    // number of image bytes required to store one stego byte

private static byte[] buildStego(String inputText)
{
    // convert data to byte arrays
    byte[] msgBytes = inputText.getBytes();
    byte[] lenBs = intToBytes(msgBytes.length);

    int totalLen = lenBs.length + msgBytes.length;
    byte[] stego = new byte[totalLen];    // for holding resulting stego

    // combine the two fields into one byte array
    System.arraycopy(lenBs, 0, stego, 0, lenBs.length);
        // length of binary message
    System.arraycopy(msgBytes, 0, stego, lenBs.length, msgBytes.length);
        // binary message

    return stego;
} // end of buildStego()

```

`buildStego()` makes use of `System.arraycopy()` to build up the byte array; its method prototype is:

```

void arraycopy(Object src, int srcPos,
               Object dest, int destPos, int length);

```

It copies the contents of `src` starting from the `srcPos` position. The data is copied into `dest` starting at the `destPos` position, and the copy is `length` bytes long.

`intToBytes()` converts its integer argument into a byte array by utilizing the fact that a Java integer is four bytes in size. Each byte of the integer is extracted, and placed in a separate cell of the array.

```

// global
private static final int MAX_INT_LEN = 4;

private static byte[] intToBytes(int i)
{
    // map the parts of the integer to a byte array
    byte[] integerBs = new byte[MAX_INT_LEN];
    integerBs[0] = (byte) ((i >>> 24) & 0xFF);
    integerBs[1] = (byte) ((i >>> 16) & 0xFF);
    integerBs[2] = (byte) ((i >>> 8) & 0xFF);
    integerBs[3] = (byte) (i & 0xFF);
    return integerBs;
} // end of intToBytes()

```

`accessBytes()` accesses the image's pixel data as a byte array, so the stego and the image are in the same byte array format.

```

private static byte[] accessBytes(BufferedImage image)
{
    WritableRaster raster = image.getRaster();
    DataBufferByte buffer = (DataBufferByte) raster.getDataBuffer();

```

```

    return buffer.getData();
} // end of accessBytes()

```

The code in `accessBytes()` can be understood by considering the data structures that make up a `BufferedImage` object, as shown in Figure 5.

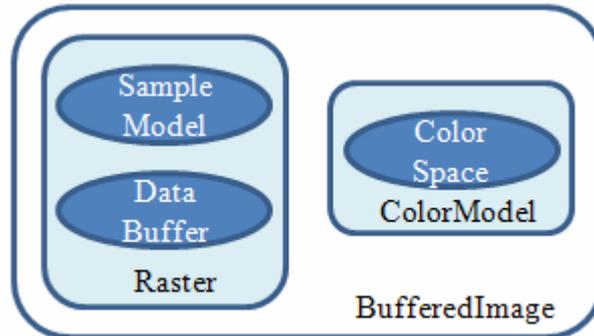


Figure 5. A `BufferedImage` Object.

The `Raster` class manages the image data, employing the `DataBuffer` class to hold the pixel data. The `SampleModel` class interprets the buffer data as pixels, and the `ColorModel` class provides a color interpretation for those pixels. `accessBytes()` accesses the raster using `WritableRaster`, which is the `Raster` subtype employed by `BufferedImage` so its pixel data can be modified.

2.2. Combining the Text and the Image

Having obtained a byte array for the stego (by calling `buildStego()`), and a byte array for the image data (by calling `accessBytes()`), the two arrays are combined as shown in Figure 3. `singleHide()` does some checking first, before calling `hideStego()` to do the necessary bit manipulation.

```

private static boolean singleHide(byte[] imBytes, byte[] stego)
{
    int imLen = imBytes.length;
    System.out.println("Byte length of image: " + imLen);

    int totalLen = stego.length;
    System.out.println("Total byte length of message: " + totalLen);

    // check that the stego will fit into the image
    /* multiply stego length by number of image bytes
       required to store one stego byte */
    if ((totalLen*DATA_SIZE) > imLen) {
        System.out.println("Image not big enough for message");
        return false;
    }

    hideStego(imBytes, stego, 0); // hide at start of image
    return true;
} // end of singleHide()

```

singleHide() checks whether the image is big enough to store the stego, bearing in mind that each bit of the stego requires one byte in the image.

hideStego() loops through the bits of the stego, modifying the LSB of each image byte, starting at the 'offset' byte position.

```
private static void hideStego(byte[] imBytes, byte[] stego,
                             int offset)
{
    for (int i = 0; i < stego.length; i++) { // loop through stego
        int byteVal = stego[i];
        for(int j=7; j >= 0; j--) { // loop through 8 bits of stego byte
            int bitVal = (byteVal >>> j) & 1;
            // change last bit of image byte to be the stego bit
            imBytes[offset] = (byte)((imBytes[offset] & 0xFE) | bitVal);
            offset++;
        }
    }
} // end of hideStego()
```

The outer for-loop iterates through the bytes of the stego, while the inner loop processes the bits of each byte. The ">>>" operator is a right shift with zero extension which, when combined with the bit-wise AND("&"), extracts bits starting from the left-hand side of the byte. The j counter of the inner loop refers to the bit indices as shown in Figure 6.

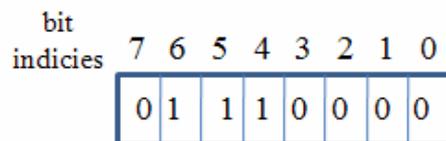


Figure 6. The Bits of a Byte.

Bits are accessed from left to right, so 0, 1, 1, 1, 0, 0, 0, and 0 would be returned from the byte in Figure 6.

Each bit is added to an image byte by combining bitwise-AND("&") and bitwise-OR("|") in:

```
imBytes[offset] & 0xFE) | bitVal
```

The bitwise-AND operation with the hexadecimal FE (1111110 binary) clears the right-most bit of imBytes[offset] (i.e. the LSB), and the bitwise-OR places the stego bit value into the empty space.

hide() finishes by calling writeImageToFile(), which saves the BufferedImage into a new file. It's not necessary to create a new BufferedImage object, since modifications to the image's byte array (i.e. imBytes[]) automatically update the loaded image because it's a WritableRaster.

2.3. Revealing a Message

The other public method in the Steganography class is:

- `boolean reveal(String imFnm);`

`reveal()` retrieves the hidden message from the `imFnm` image file, storing it in `<imFnm>.txt`

```
public static boolean reveal(String imFnm)
{
    // get the image's data as a byte array
    BufferedImage im = loadImage(imFnm);
    if (im == null)
        return false;
    byte[] imBytes = accessBytes(im);
    System.out.println("Byte length of image: " + imBytes.length);

    // get msg length at the start of the image
    int msgLen = getMsgLength(imBytes, 0);
    if (msgLen == -1)
        return false;
    System.out.println("Byte length of message: " + msgLen);

    // get message located after the length info in the image
    String msg = getMessage(imBytes, msgLen, MAX_INT_LEN*DATA_SIZE);
    if (msg != null) { // save message in a text file
        String fnm = getFileName(imFnm);
        return writeStringToFile(fnm + ".txt", msg);
    }
    else {
        System.out.println("No message found");
        return false;
    }
} // end of reveal()
```

The image data is accessed in the same way as in `hide()`: first a `BufferedImage` is created with `loadImage()`, and its data accessed as a byte array by `accessBytes()`.

Hidden message extraction is done in two stages – first the size of the binary message is read from the image bytes with `getMsgLength()`, then this length is used to constrain the reading of the rest of the image by `getMessage()` so that only the message is retrieved.

`getMsgLength()` retrieves a byte array of size `MAX_INT_LEN` (4) first, by calling `extractHiddenBytes()`, and then converting the resulting byte array to a Java integer.

```
private static int getMsgLength(byte[] imBytes, int offset)
{
    byte[] lenBytes = extractHiddenBytes(imBytes, MAX_INT_LEN, offset);
    // get the binary message length as a byte array
    if (lenBytes == null)
        return -1;

    // convert the byte array into an integer
    int msgLen = ((lenBytes[0] & 0xff) << 24) |
                ((lenBytes[1] & 0xff) << 16) |
                ((lenBytes[2] & 0xff) << 8) |
                (lenBytes[3] & 0xff);
}
```

```

    if ((msgLen <= 0) || (msgLen > imBytes.length)) {
        System.out.println("Incorrect message length");
        return -1;
    }
    return msgLen;
} // end of getMsgLength()

```

`extractHiddenBytes()` extracts a specified number of bytes from the image data, starting at the 'offset' position.

```

private static byte[] extractHiddenBytes(byte[] imBytes,
                                         int size, int offset)
{
    int finalPosn = offset + (size*DATA_SIZE);
    if (finalPosn > imBytes.length) {
        System.out.println("End of image reached");
        return null;
    }

    byte[] hiddenBytes = new byte[size];

    for (int j = 0; j < size; j++) { // loop through hidden bytes
        for (int i=0; i < DATA_SIZE; i++) {
            // make one hidden byte from DATA_SIZE image bytes
            hiddenBytes[j] = (byte) ((hiddenBytes[j] << 1) |
                                     (imBytes[offset] & 1));

            /* shift existing bits left;
               store LSB of image byte on the right */
            offset++;
        }
    }
    return hiddenBytes;
} // end of extractHiddenBytes()

```

`extractHiddenBytes()` begins with some size-checking to ensure that there's enough data left to extract. The test uses the fact that each byte of the stego is stored in `DATA_SIZE` (8) bytes of image data, so if 'size' bytes of stego need to be retrieved, then the remaining image data must equal or exceed `size*DATA_SIZE` bytes.

A `hiddenBytes[]` byte array is created to hold the bytes of the extracted text message. A nested loop is entered, where the outer loop fills `hiddenBytes[]` with bytes, and the inner loop constructs a `hiddenBytes[]` byte a bit at a time.

The LSB of the image data is read using bitwise-AND:

```
imBytes[offset] & 1)
```

The extracted bit is added to the right end of the `hiddenBytes[j]` byte with a bitwise-OR, *after* the existing contents of the byte have been shifted one bit to the left with "`<<`":

```
hiddenBytes[j] = (byte) ((hiddenBytes[j] << 1) |
                          (imBytes[offset] & 1));
```

The bit operations are illustrated by Figure 7.

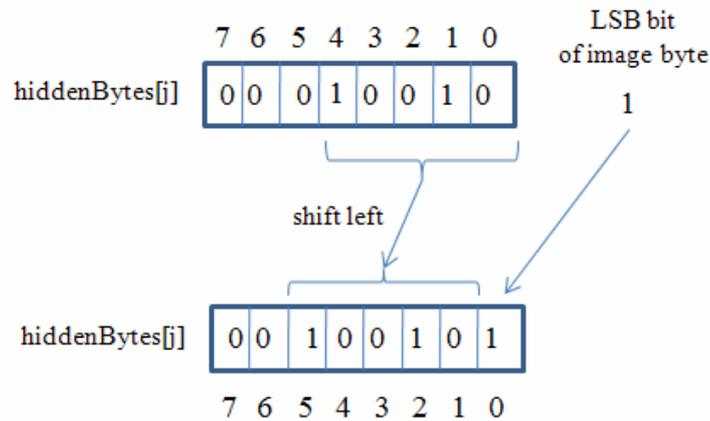


Figure 7. Constructing a hiddenBytes[] Byte.

The existing bits in hiddenBytes[j] (10010) are shifted to the left before the image's LSB is inserted on the right.

Once the length of the stego message is known, the rest of the image data is scanned by extractHiddenBytes() in getMessage().

```
private static String getMessage(byte[] imBytes, int msgLen, int
offset)
{
    byte[] msgBytes = extractHiddenBytes(imBytes, msgLen, offset);
    // the message is msgLen bytes long
    if (msgBytes == null)
        return null;

    String msg = new String(msgBytes);

    // check the message is all characters
    if (isPrintable(msg))
        return msg;
    else
        return null;
} // end of getMessage()
```

extractHiddenBytes() extracts msgLen bytes, the number of bytes in the message.

The byte array is converted to a string, and checked to see if it consists of printable characters. If the message has been corrupted, for example by someone cropping or scaling the image, then it's very likely that some of the message will be unreadable.

2.4. Evaluating the Steganography Class

The principle weakness of the Steganography class is that it uses an extremely well-known technique for hiding a message. Probably every steganalysis tool supports LSB search. One answer is to vary the encoding technique, such as by storing a message bit in a different part of an image byte.

Another solution is to keep using LSB, since it's easy to implement, but encrypt the message before hiding it in the image. A steganalysis tool will still find the hidden encrypted bits, but they'll look like random data rather a text message.

3. The StegCrypt Class

StegCrypt is a modification of the Steganography class which encrypts the binary version of the plain-text message before hiding it in an image.

The byte array encryption and decryption is done using Jasypt's BasicBinaryEncryptor class. Jasypt (<http://www.jasypt.org/>) lets a programmer employ basic encryption capabilities without requiring a deep knowledge of cryptography.

Jasypt doesn't implement any encryption algorithms itself, but acts as a simplifying interface to the JCE (Java Cryptography Extension), an API that provides security features via the JVM (or third party providers such as Bouncy Castle (<http://www.bouncycastle.org/>)).

A fundamental question with encryption is how to safely send the encrypted message's password (or key) to the message's recipient. I 'solve' this problem by including the password as part of the stego message, which now has three fields:

```
<password>
<size of encrypted binary message>
<encrypted binary message>
```

This approach is fine since the aim of the encryption is only to scramble the message so that a basic LSB steganalysis attack will not reveal plain-text. Of course, if the attacker knows that the first field is the message's password, then the original message can be retrieved.

The password is a 10-letter randomly generated string, which will be stored in 10*8 bytes in the image. The message size is a Java integer (i.e. 4 bytes long), which utilizes 4*8 bytes in the image. Each byte of the encrypted message needs 8 bytes of storage in the image.

The StegCrypt class has the same public interface as Steganography, two public static methods:

- `boolean hide(String textFnm, String imFnm);`
- `boolean reveal(String imFnm);`

3.1. Hiding a Message

hide() loads the text and image, and saves the modified image in the same way as the Steganography class. The differences come in the use of a password to encrypt the text:

```
public static boolean hide(String textFnm, String imFnm)
/* hide message read from textFnm inside image read from imFnm;
```

```

    the resulting image is stored in <inFnm>Msg.png */
{
    // read in the message as a byte array
    byte[] msgBytes = readMsgBytes(textFnm);
    if (msgBytes == null)
        return false;

    // generate a password
    String password = genPassword();
    byte[] passBytes = password.getBytes();

    // use password to encrypt the message
    byte[] encryptedMsgBytes = encryptMsgBytes(msgBytes, password);
    if (encryptedMsgBytes == null)
        return false;

    byte[] stego = buildStego(passBytes, encryptedMsgBytes);

    // access the image's data as a byte array
    BufferedImage im = loadImage(imFnm);
    if (im == null)
        return false;
    byte imBytes[] = accessBytes(im);

    if (!singleHide(imBytes, stego)) // im is modified with the stego
        return false;

    // store the modified image in <fnm>Msg.png
    String fnm = getFileName(imFnm);
    return writeImageToFile( fnm + "Msg.png", im);
} // end of hide()

```

One benefit of staying with LSB hiding, is that once the byte array holding the three-part stego message has been constructed (i.e. the stego[] array is filled), then the same singleHide() method as in Steganography can be employed to write it into the image's data.

genPassword() generates a 10-letter random string.

```

// globals
private static final int PASSWORD_LEN = 10;
private static Random rand = new Random(); //used for passwd gen

private static String genPassword()
{
    String availChars =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ23456789";
        // chars available for password

    StringBuffer sb = new StringBuffer(PASSWORD_LEN);
    for (int i=0; i<PASSWORD_LEN; i++ ) {
        int pos = rand.nextInt( availChars.length());
        sb.append( availChars.charAt(pos));
    }
    return sb.toString();
} // end of genPassword()

```

It's important that the password isn't a recognizable word, since this would make it easier to spot during steganalysis.

The encryption employs Jasypt's `BasicBinaryEncryptor` class, based upon an example in the code samples at <http://www.jasypt.org/getting-started-easy.html>

```
private static byte[] encryptMsgBytes(byte[] msgBytes,
                                     String password)
// encrypt a message (a byte array) using the password
{
    BasicBinaryEncryptor bbe = new BasicBinaryEncryptor();
    bbe.setPassword(password);
    return bbe.encrypt(msgBytes);
}
```

`buildStego()` creates a byte array containing the three fields (the password, the size of encrypted binary message, and the encrypted binary message). As in `buildStego()` in the `Steganography` class, the coding employs `System.arraycopy()`:

```
private static byte[] buildStego(byte[] passBytes,
                                byte[] encryptedMsgBytes)
{
    byte[] lenBs = intToBytes(encryptedMsgBytes.length);

    int totalLen = passBytes.length + lenBs.length +
                  encryptedMsgBytes.length;
    byte[] stego = new byte[totalLen];    // for holding the stego

    // combine the 3 fields into one byte array
    int destPos = 0;
    System.arraycopy(passBytes, 0, stego, destPos, passBytes.length);
    destPos += passBytes.length;    // add the password

    System.arraycopy(lenBs, 0, stego, destPos, lenBs.length);
    destPos += lenBs.length;    //add length of encrypted binary message

    System.arraycopy(encryptedMsgBytes, 0, stego, destPos,
                    encryptedMsgBytes.length);
    // encrypted binary message

    return stego;
} // end of buildStego()
```

3.2. Revealing a Message

`StegCrypt`'s `reveal()` must divide the hidden data into three parts, and decrypt the message segment.

```
public static boolean reveal(String imFnm)
/* Retrieve the hidden message from imFnm from the beginning
   of the image. */
{
    // get the image's data as a byte array
    BufferedImage im = loadImage(imFnm);
    if (im == null)
```

```

    return false;
    byte[] imBytes = accessBytes(im);
    int imLen = imBytes.length;
    System.out.println("Byte Length of image: " + imLen);

    String msg = extractMsg(imBytes, 0);
    if (msg != null) { // save message in a text file
        String fnm = getFileName(imFnm);
        return writeStringToFile(fnm + ".txt", msg);
    }
    else {
        System.out.println("No message found");
        return false;
    }
} // end of reveal()

```

The changed code, compared to `reveal()` in the `Steganography` class, is located in `extractMsg()`. It extracts the password and encrypted binary message length, and uses them to reconstruct the hidden message.

```

private static String extractMsg(byte[] imBytes, int offset)
{
    String password = getPassword(imBytes, offset);
    if (password == null)
        return null;

    offset += PASSWORD_LEN*DATA_SIZE; // move past password
    int msgLen = getMsgLength(imBytes, offset);
    if (msgLen == -1)
        return null;

    offset += MAX_INT_LEN*DATA_SIZE; // move past message length
    return getMessage(imBytes, msgLen, password, offset);
} // end of extractMsg()

```

`getPassword()` retrieves the password from the image using `extractHiddenBytes()` which I described in the `Steganography` class.

```

private static String getPassword(byte[] imBytes, int offset)
{
    byte[] passBytes =
        extractHiddenBytes(imBytes, PASSWORD_LEN, offset);
    if (passBytes == null)
        return null;
    String password = new String(passBytes);

    // check the password is all characters
    if (isPrintable(password))
        return password;
    else
        return null;
} // end of getPassword()

```

It uses `isPrintable()` to check that the string hasn't been corrupted.

`getMessage()` pulls the encrypted binary message from the image, decrypts it with the password, and converts it to a string.

```
private static String getMessage(byte[] imBytes,
                                int msgLen, String password, int offset)
{
    byte[] enMsgBytes = extractHiddenBytes(imBytes, msgLen, offset);
    // the encrypted message is msgLen bytes long
    if (enMsgBytes == null)
        return null;

    // decrypt the message using the password
    BasicBinaryEncryptor bbe = new BasicBinaryEncryptor();
    bbe.setPassword(password);
    byte[] msgBytes = null;
    try {
        msgBytes = bbe.decrypt(enMsgBytes);
    }
    catch(Exception e) // in case the decryption fails
    { System.out.println("Problem decrypting message");
      return null;
    }

    String msg = new String(msgBytes);
    if (isPrintable(msg)) // check the message is all chars
        return msg;
    else
        return null;
} // end of getMessage()
```

The decryption utilizes Jasypt's `BasicBinaryEncryptor` class. The decryption may fail if some part of the message has been corrupted inside the image.

3.3. Evaluating the StegCrypt Class

A LSB steganalysis of an image modified by `StegCrypt` returns an unintelligible stream of binary, although the password string is visible at the start. Nevertheless, an eagle-eyed investigator will still be able to detect the message if the original, unmodified image is available for comparison. As Figure 1 indicates, LSB steganography isn't noticeable to the human eye, even when the two versions of an image are placed side-by-side. However, image analysis tools *can* detect a difference.

To illustrate the point, I'll utilize `ImageJ` (<http://rsbweb.nih.gov/ij/>), an open source Java image processing application, which can analyze numerous image formats, including PNG. It supports a wide range of operations useful for steganalysis, such as pixel value statistics, density histograms, and line profile plots. It also offers standard image processing functions such as contrast manipulation, sharpening, smoothing, geometric transformations, edge detection and filtering. `ImageJ`'s functionality can be extended with plug-ins, with over 500 currently available.

If the original image, and a version containing a hidden message, are loaded into `ImageJ`, their differences can be visualized using `ImageJ`'s image calculator, invoked as in Figure 8.

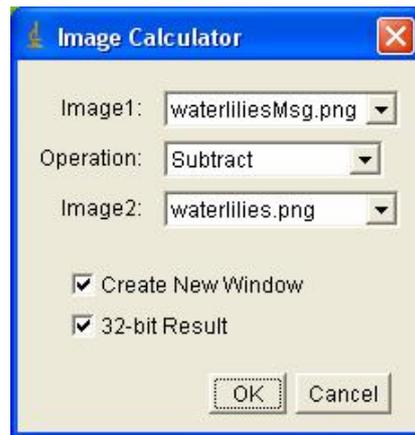


Figure 8. Subtracting Two Images in ImageJ.

waterliliesMsg.png is an image containing a hidden, encrypted message, while waterlilies.png is the original. Subtraction produces a new image, which is partly shown in Figure 9, enlarged 6 times.

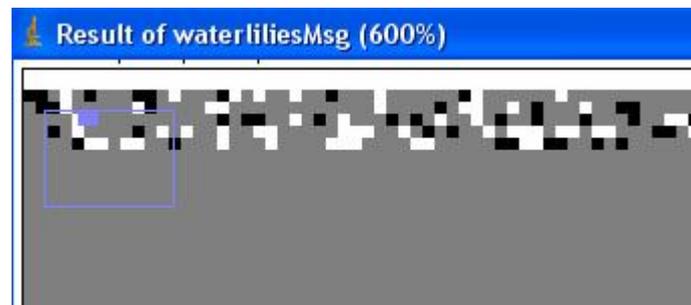


Figure 9. The Result of Subtracting Two Images.

The top few rows of the image contain a scattering of non-grey pixels, which indicates a difference between the images. Although we don't know what the hidden message says, we have found it and, with a small amount of cropping, the message can be deleted.

The subtraction can be visualized in ImageJ as a wire-framed surface plot, shown in Figure 10.

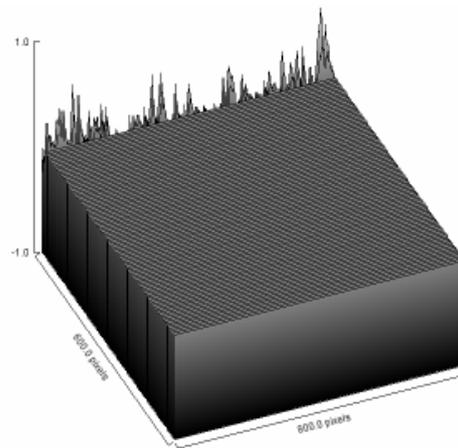


Figure 10. A Surface Plot of the Subtracted StegCrypt Image.

The message stands out as variations from 0, along the first few rows.

Although the difference between the two images seems to be just noise, its presence *only* at the start of the image raises suspicions, and makes it easy to delete.

My solution, detailed in the next section, is to insert the steganographic message into the image *multiple* times –spreading the 'noise' throughout the image, and making it harder to remove the message by cropping.

4. The MultiStegCrypt Class

The MultiSteg stores multiple copies of the stego message in the image, inserting them one after another from the beginning of the file until there's no room for another complete copy.

It's useful to add a header string to the start of each copy. After image cropping, the headers make it easier to search the image and identify a surviving message.

Each stego message now has four fields:

```
"XXXXXX"
<password>
<size of encrypted binary message>
<encrypted binary message>
```

The "XXXXXX" string is the header. The password, size, and binary message are the same as in StegCrypt, with the encryption and decryption of byte arrays achieved with Jasypt's BasicBinaryEncryptor class.

4.1. Hiding the Message

hide() is largely unchanged from previously, except that buildStego() adds the header string to the front of the stego byte array, and singleHide() is replaced by mutipleHide().

```

public static boolean hide(String textFnm, String imFnm)
{
    // read in the message as a byte array
    byte[] msgBytes = readMsgBytes(textFnm);
    if (msgBytes == null)
        return false;

    // generate a password
    String password = genPassword();
    byte[] passBytes = password.getBytes();

    // use password to encrypt the message
    byte[] encryptedMsgBytes = encryptMsgBytes(msgBytes, password);
    if (encryptedMsgBytes == null)
        return false;

    byte[] stego = buildStego(passBytes, encryptedMsgBytes);

    // access the image's data as a byte array
    BufferedImage im = loadImage(imFnm);
    if (im == null)
        return false;
    byte imBytes[] = accessBytes(im);

    // im is modified with multiple copies of stego
    if (!multipleHide(imBytes, stego))
        return false;

    // store the modified image in <fnm>Msg.png
    String fnm = getFileName(imFnm);
    return writeImageToFile( fnm + "Msg.png", im);
} // end of hide()

```

buildStego() adds the "XXXXXX" header to the front of the stego message by utilizing `System.arraycopy()`:

```

// globals
private static final String STEGO_HEADER = "XXXXXX";

private static byte[] buildStego(byte[] passBytes,
                                byte[] encryptedMsgBytes)
{
    // create two of the fields (as byte arrays)
    byte headerBytes[] = STEGO_HEADER.getBytes(); // "XXXXXX"
    byte[] lenBs = intToBytes(encryptedMsgBytes.length);

    int totalLen = STEGO_HEADER.length() + passBytes.length +
                  lenBs.length + encryptedMsgBytes.length;
    byte[] stego = new byte[totalLen]; // for holding the stego

    // combine the four fields into one byte array
    int destPos = 0;
    System.arraycopy(headerBytes, 0, stego,
                    destPos, STEGO_HEADER.length()); // header
    destPos += STEGO_HEADER.length();
    System.arraycopy(passBytes, 0, stego,
                    destPos, passBytes.length); // password
    destPos += passBytes.length;
}

```

```

    System.arraycopy(lenBs, 0, stego,
                    destPos, lenBs.length);    // length of message
    destPos += lenBs.length;
    System.arraycopy(encryptedMsgBytes, 0, stego,
                    destPos, encryptedMsgBytes.length); //message
    return stego;
} // end of buildStego()

```

Whereas the old `singleHide()` called `hideStego()` once to add the message to the image, `multipleHide()` calls `hideStego()` inside a loop. Each `hideStego()` call inserts a complete copy of the message, and so usually the very end of the image is left unmodified because there's not enough space left to add a complete message.

```

private static boolean multipleHide(byte[] imBytes, byte[] stego)
{
    int imLen = imBytes.length;
    System.out.println("Byte length of image: " + imLen);

    int totalLen = stego.length;
    System.out.println("Total byte length of message: " + totalLen);

    //check that the stego will fit into the image
    /* multiply stego length by number of image bytes required
       to store one stego byte */
    if ((totalLen*DATA_SIZE) > imLen) {
        System.out.println("Image not big enough for message");
        return false;
    }

    // calculate the number of times the stego can be hidden
    int numHides = imLen/(totalLen*DATA_SIZE);    // integer div
    System.out.println("No. of message duplications: " + numHides);

    for(int i=0; i < numHides; i++)    // hide stego numHides times
        hideStego(imBytes, stego, (i*totalLen*DATA_SIZE));

    return true;
} // end of multipleHide()

```

4.2. Revealing the Message

The payoff of multiple hides is that `reveal()` needn't just give up if a stego message has been corrupted, it can keep searching for another copy in the image.

```

public static boolean reveal(String imFnm)
{
    // get the image's data as a byte array
    BufferedImage im = loadImage(imFnm);
    if (im == null)
        return false;
    byte[] imBytes = accessBytes(im);

    int imLen = imBytes.length;
    System.out.println("Byte Length of image: " + imLen);

    int headOffset = STEGO_HEADER.length()*DATA_SIZE;
    // stego header space used in image

```

```

String msg = null;
boolean foundMsg = false;
int i = 0;
while ((i < imLen) && !foundMsg) {
    if (!findHeader(imBytes, i)) // no stego header found at pos i
        i++; // move on
    else { // found header
        i += headOffset; // move past stego header
        msg = extractMsg(imBytes, i);
        if (msg != null)
            foundMsg = true;
    }
}

if (foundMsg) {
    String fnm = getFileName(imFnm);
    return writeStringToFile(fnm + ".txt", msg);
}
else {
    System.out.println("No message found");
    return false;
}
} // end of reveal()

```

reveal()'s while-loop moves through the image looking for a stego header. When one is found, extractMsg() tries to retrieve the rest of the message. If the extraction is successful, reveal() can finish, otherwise the search continues until a message is found or the end of the image is reached.

findHeader() relies on extractHiddenBytes() to pull out the LSBs of the image data and store them in a byte array. The array is converted to a string and compared to the stego header ("XXXXXX").

```

private static boolean findHeader(byte[] imBytes, int offset)
// does a stego header start at the offset position in the image?
{
    byte[] headerBytes =
        extractHiddenBytes(imBytes, STEGO_HEADER.length(), offset);
    if (headerBytes == null)
        return false;

    String header = new String(headerBytes);
    if (!header.equals(STEGO_HEADER))
        return false;

    return true;
} // end of findHeader()

```

4.3. Evaluating the MultiStegCrypt Class

One of the aims of MultiStegCrypt is to repeat the message throughout the image so image analysis won't highlight localized changes (as illustrated in Figure 10).

When the ImageJ's image subtraction is applied to an image modified using MultiStegCrypt, the surface plot looks like Figure 11.

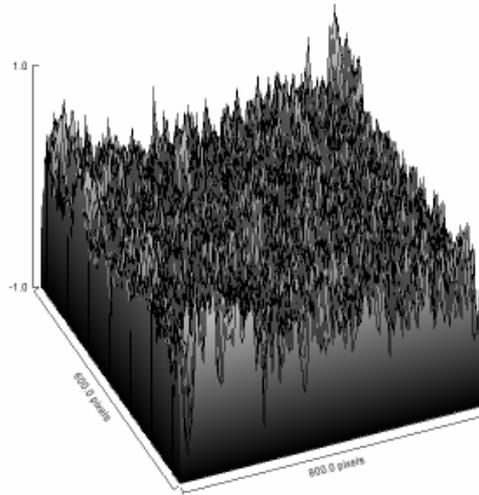


Figure 11. A Surface Plot of the Subtracted MultiStegCrypt Image.

The data appears to be random throughout, although a very close look at the end of the image shows that the last row and a half of the subtracted image is uniformly gray. The very last part of the image was not modified because there wasn't enough space to add a complete stego message.

Another aim of MultiStegCrypt is to make the hidden message retrievable in the presence of cropping. Unfortunately, MultiStegCrypt isn't very successful at that task.

If an image is cropped along the y-axis so that, for example, its lower half is removed as in Figure 12, then the message can still be found.



Figure 12. Cropped Water Lilies along the Y-Axis (Message Found).

But if the image is cropped along the x-axis so the right side of the image is removed as in Figure 13, then the message is lost.



Figure 13. Cropped Water Lilies along the X-Axis (Message Lost).

This behavior can be understood by considering how the message is added to the picture: copies are inserted one after another, horizontally across the image, row-by-

row. Also, each message is quite large – the header requires 5 bytes for the "XXXXX", 10 bytes for the password, and 4 bytes for the message length (a Java integer). That's a total of 19 bytes, which requires 152 (19*8) bytes in the image, and that's before we consider that each byte of the text needs 8 bytes of storage.

As a consequence, even a small piece of hidden text will frequently use more than one row of image bytes. If the image is cropped along the x-axis, as in Figure 13, then it's almost certain that *every* copy of the message will be affected since every copy is stored over more than one image row.

My solution is to switch from storing multiple copies of the entire message, to storing multiple *fragments*. The message is split up into small pieces, and repeatedly written into the image. The storage of smaller chunks means that cropping is less likely to destroy so many that it's no longer possible to reconstitute the original message.

5. The FragMultiStegCrypt Class

The FragMultiStegCrypt class is a version of MultiStegCrypt class that stores multiple fragments of the stego message. Each fragment has five fields:

```
"XXX" <fragment number>
<password>
<size of encrypted binary message fragment>
<encrypted binary message fragment>
```

"XXX" is the stego fragment header, reduced from "XXXXX" to save on space.

There's always exactly 100 fragments, and so the fragment number range between 0 and 99. This means it can be encoded as a string of two characters, requiring only 2 bytes, rather than as a full-blown integer of 4 bytes.

The password string is shortened to save space, down to a 3-letter random string, and each message fragment is assigned its own password. The message fragment size is still a Java integer, and each message fragment is encrypted using Jasypt.

5.1. Hiding the Message

hide() fills a 100-element array with byte arrays, which hold the 100 fragments of the encoded hidden message. Each fragment is hidden multiple times inside the image.

```
// globals
private static final int NUM_FRAGS = 100;
    // number of stego fragments for a message

public static boolean hide(String textFnm, String imFnm)
{
    /* read in message as a series of byte array fragments
       stored in an array */
    byte[][] msgFrgs = readByteFrgs(textFnm);
    if (msgFrgs == null)
        return false;

    byte[][] stegoFrgs = new byte[NUM_FRAGS][]; // holds stego frags
```

```

for (int i=0; i < NUM_FRAGS; i++) {
    String password = genPassword(); //a password for each fragment
    byte[] passBytes = password.getBytes();

    // use the password to encrypt a message fragment
    byte[] encryptedFrag = encryptFrag(msgFrag[i], password);
    if (encryptedFrag == null)
        return false;

    // create the stego fragment
    stegoFrag[i] = buildStegoFrag(i, passBytes, encryptedFrag);
}

// access the image's data as a byte array
BufferedImage im = loadImage(imFnm);
if (im == null)
    return false;
byte imBytes[] = accessBytes(im);

// im is modified with multiple copies of the stego frags
if (!multipleHide(imBytes, stegoFrag))
    return false;

// store the modified image in <fnm>Msg.png
String fnm = getFileName(imFnm);
return writeImageToFile( fnm + "Msg.png", im);
} // end of hide()

```

In all previous versions of `hide()`, the message has been read in as a single byte array, but in `FragMultiStegCrypt`, it's stored as a series of byte arrays in `msgFrag[][]`. Inside the for-loop, each fragment is encrypted and added to another array of byte arrays called `stegoFrag[][]`. This array is passed to `multipleHide()` which hides each encrypted fragment multiple times inside the image.

`readByteFrag()` starts by reading in a text file as a single string, but then moves through it chopping it into substrings which are stored as byte arrays in `msgFrag[][]`.

```

private static byte[][] readByteFrag(String fnm)
{
    String inputText = readTextFile(fnm);
    if ((inputText == null) || (inputText.length() == 0))
        return null;

    // decide on fragment length
    int fragLen = inputText.length() / NUM_FRAGS; // integer div
    System.out.println("Input text size: " + inputText.length() +
        "; fragment length: " + fragLen);

    if (fragLen == 0) { // since text length < NUM_FRAGS
        inputText = String.format("%1$-" + NUM_FRAGS + "s", inputText);
        // right pad the text with spaces so NUM_FRAGS long
        fragLen = 1;
    }

    // split the input text into multiple fragments (byte arrays)
    int startPosn = 0;
    int endPosn = fragLen;
    String textFrag;
    byte[][] msgFrag = new byte[NUM_FRAGS][];

```

```

for(int i=0; i < NUM_FRAGS-1; i++) {
    textFrag = inputText.substring(startPosn, endPosn); //cut out frag
    msgFrag[i] = textFrag.getBytes();
    startPosn = endPosn;
    endPosn += fragLen;
}

textFrag = inputText.substring(startPosn); // store remaining frag
msgFrag[NUM_FRAGS-1] = textFrag.getBytes();

return msgFrag;
} // end of readByteFrag()

```

FragMultiStegCrypt is made less complicated by assuming there's always 100 (NUM_FRAGS) fragments. But there's a problem if the input text has less than NUM_FRAGS characters, and so can't be subdivided. My answer is to pad short strings with enough spaces to make them NUM_FRAGS characters long. This is done using String.format():

```
inputText = String.format("%1$-" + NUM_FRAGS + "s", inputText);
```

The format string is processed as "%1\$-100s" which generates a 100-char long string, padded with spaces after the input text (which is referenced using "1\$"). The "-" left justifies the output so the padding is added on the right. A string of length NUM_FRAGS can then be split into NUM_FRAG fragments, each 1 character long.

5.2. Building a Fragment

buildStegoFrag() builds a single stego fragment (a byte array), made up of the five fields mentioned at the start of the section. As with previous versions of this code, the byte array is constructed using calls to System.arraycopy() which gradually fill in the array with the fields.

```

private static byte[] buildStegoFrag(int i, byte[] passBytes,
                                     byte[] encryptedFrag)
/* Build a single stego fragment (a byte array), made up of 5 fields:
   "XXX" <fragment number> <password>
   <size of encrypted binary message fragment>
   <encrypted binary message fragment>
*/
{ // create three of the fields (as byte arrays)
    byte headerBytes[] = STEGO_HEADER.getBytes(); // "XXX"
    byte[] fragNumBs = fragNumberToBytes(i);
    byte[] lenBs = intToBytes(encryptedFrag.length);

    int totalLen = STEGO_HEADER.length() + fragNumBs.length +
                   passBytes.length + lenBs.length +
                   encryptedFrag.length;

    byte[] sFrag = new byte[totalLen]; // for holding stego fragment

    // combine the five fields into one byte array
    int destPos = 0;
    System.arraycopy(headerBytes, 0, sFrag, destPos,
                     STEGO_HEADER.length()); // header

```

```

destPos += STEGO_HEADER.length(); // fragment number
System.arraycopy(fragNumBs, 0, sFrag, destPos, fragNumBs.length);

destPos += fragNumBs.length; // password
System.arraycopy(passBytes, 0, sFrag, destPos, passBytes.length);

destPos += passBytes.length;
System.arraycopy(lenBs, 0, sFrag, destPos, lenBs.length);
// length of encrypted binary message

destPos += lenBs.length;
System.arraycopy(encryptedFrag, 0, sFrag, destPos,
                encryptedFrag.length); // encrypted binary message

return sFrag;
} // end of buildStegoFrag()

```

5.3. Hiding Fragments

multipleHide() is more complicated than previously since it has to store stego fragments multiple times inside the image.

The trickiest aspect of multiple insertion is deciding how many times each fragment should be inserted (the numHides value in the code). I went for an easy solution by dividing the total size of all the fragments into the size of the image to get numHides.

```

private static boolean multipleHide(byte[] imBytes,
                                   byte[][] stegoFragments)
// store the stego fragments multiple times in the image
{
    int imLen = imBytes.length;
    System.out.println("Byte length of image: " + imLen);

    // calculate total length of all the stego fragments
    int totalLen = 0;
    for(int i=0; i < NUM_FRAGS; i++)
        totalLen += stegoFragments[i].length;
    System.out.println("Total byte length of info: " + totalLen);

    // check that the stego will fit into the image
    /* multiply stego length by number of image bytes required
       to store one stego byte */
    if ((totalLen*DATA_SIZE) > imLen) {
        System.out.println("Image not big enough for message");
        return false;
    }

    // calculate number of times the complete stego can be hidden
    int numHides = imLen/(totalLen*DATA_SIZE); // integer div
    System.out.println("No. of message duplications: " + numHides);

    int offset = 0;
    for(int h=0; h < numHides; h++) //hide all frags, numHides times
        for(int i=0; i < NUM_FRAGS; i++) {
            hideStegoFrag(imBytes, stegoFragments[i], offset);
            offset += stegoFragments[i].length*DATA_SIZE;
        }
}

```

```

    return true;
} // end of multipleHide()

```

Fragment insertion is coded with nested for-loops which call `hideStegFrag()` to store a fragment in the LSBs of the image's bytes. This means that a copy of a message is only added to the image if there's room to insert all of its fragments. The drawback is that there will usually be some unused space at the end of the image where some extra fragments could have been stored.

5.4. Revealing the Message

`reveal()` searches through the image gathering stego fragments. If the image has been cropped then a given fragment may be incomplete. This won't stop the search, which will scan the entire image, and hopefully find missing fragments somewhere else.

The search stops either when all the stego fragments have been found, or when the end of the file is reached.

Even if some of the stego fragments are still missing by the end of the search, the other fragments are decrypted, and the partial text saved in a file.

```

public static boolean reveal(String imFnm)
{
    // access the image's data as a byte array
    BufferedImage im = loadImage(imFnm);
    if (im == null)
        return false;
    byte[] imBytes = accessBytes(im);

    String[] msgFrgs = new String[NUM_FRAGS]; // holds message frags
    int numFrgsFound = findFragments(imBytes, msgFrgs);
    if (numFrgsFound == 0) {
        System.out.println("No message found");
        return false;
    }

    String msg = combineFragments(msgFrgs, numFrgsFound);
    String fnm = getFileName(imFnm);
    return writeStringToFile(fnm + ".txt", msg); // save message
} // end of reveal()

```

5.5. Finding Fragments

`findFragments()` retrieves the message fragments from the image, storing them in an array called `msgFrgs[]`, and returns the number of fragments found. It only has to search the image until it has found all `NUM_FRAG` fragments, and it doesn't need to decrypt a fragment again once it's been found.

```

private static int findFragments(byte[] imBytes, String[] msgFrgs)
{
    for(int i=0; i < NUM_FRAGS; i++)
        msgFrgs[i] = null;
    int numFrgsFound = 0;

```

```

int imLen = imBytes.length;
System.out.println("Byte Length of image: " + imLen);

int headOffset = STEGO_HEADER.length()*DATA_SIZE;
                // stego header space used in image
int fragNumOffset = NUM_FRAG_LEN*DATA_SIZE;
                // fragment number space used in image

int i = 0;
while ((i < imLen) && (numFragmentsFound < NUM_FRAGS)) {
    if (!findHeader(imBytes, i)) // no stego header found at pos i
        i++; // move on
    else { // found header
        i += headOffset; // move past stego header
        int fNum = findFragNumber(imBytes, i);
        if (fNum != -1) { // found a fragment number
            i += fragNumOffset; // move past fragment number
            if (msgFragments[fNum] != null)
                // we already have a message for this fragment
                System.out.println("Fragment " + fNum + " already extracted");
            else { // this is a fragment we haven't seen before
                String msgFrag = extractMsgFrag(imBytes, i);
                if (msgFrag == null)
                    System.out.println("Failed to extract fragment " + fNum);
                else { // got the message fragment
                    System.out.println("Storing fragment " + fNum);
                    msgFragments[fNum] = msgFrag;
                    numFragmentsFound++;
                    i += (PASSWORD_LEN + MAX_INT_LEN +
                        currMsgFragLen)*DATA_SIZE;
                    /* move past password, length & message fragment;
                       currMsgFragLen is assigned in extractMsgFrag() */
                }
            }
        }
    }
}
return numFragmentsFound;
} // end of findFragments()

```

findFragments() is somewhat convoluted because it has to extract five fields for each fragment – the fragment header, the fragment number, the password, the size of encrypted binary message fragment, and the message fragment itself. If any of these extraction steps fails then findFragments() moves on and start searching for another fragment.

findHeader() deals with header extraction, findFragNumber() with the fragment number, and extractMsgFrag() with the password, size, and message.

To simplify the 'moving on' steps, the byte sizes of the header and the fragment number in the image are pre-calculated (and stored in headOffset and fragNumOffset). The size of the password and fragment message size fields are also fixed. The only part of the fragment which can vary in size is the one holding the actual hidden message. It's length is calculated in extractMsgFrag(), and stored in a global variable called currMsgFragLen so it can be used by findFragments().

When `findFragments()` extracts a message from a new fragment, it's stored in the `msgFragments[]` array, and the number of found fragments is incremented. This counter is used to detect when all `NUM_FRAGS` fragments have been retrieved.

If some fragments cannot be successfully extracted then some elements in `msgFragments[]` will be null, and `numFragmentsFound` will be less than `NUM_FRAGS`.

`findHeader()` is the same as the same-named method in `MultiStegCrypt`, so won't be described again. `findFragNumber()` extracts the bytes of the fragment number using `extractHiddenBytes()`, and attempts to convert them into an integer by parsing the string version of the bytes array.

```
private static int findFragNumber(byte[] imBytes, int offset)
// extract a fragment number in the range 0-(NUM_FRAGS-1)
{
    byte[] fNumBytes =
        extractHiddenBytes(imBytes, NUM_FRAG_LEN, offset);
    if (fNumBytes == null)
        return -1;

    //convert the fragment number bytes into an integer
    String fNumStr = new String(fNumBytes).trim();
    int fNum = -1;
    try {
        fNum = Integer.parseInt(fNumStr);
    }
    catch (NumberFormatException e)
    { System.out.println("Could not parse frag no: " + fNumStr); }

    if ((fNum < 0) || (fNum >= NUM_FRAGS)){
        System.out.println("Incorrect fragment number");
        return -1;
    }
    return fNum;
} // end of findFragNumber()
```

`extractMsgFrag()` does almost the same tasks as `extractMsg()` in `MultiStegCrypt`: it retrieves the hidden message fragment from the image by first extracting its password and the encrypted binary message fragment length.

```
// global
private static int currMsgFragLen;

private static String extractMsgFrag(byte[] imBytes, int offset)
{
    String password = getPassword(imBytes, offset);
    if (password == null)
        return null;

    offset += PASSWORD_LEN*DATA_SIZE; // move past password
    int msgFragLen = getMsgFragLength(imBytes, offset);
        // get the encrypted message fragment length
    if (msgFragLen == -1)
        return null;
```

```

currMsgFragLen = msgFragLen; // used by findFragments()

offset += MAX_INT_LEN*DATA_SIZE; // move past message length
return getMessageFrag(imBytes, msgFragLen, password, offset);
} // end of extractMsgFrag()

```

The only difference is that `extractMsgFrag()` assigns the message length to the global variable `currMsgFragLen`, so `findFragments()` can use it for stepping through the image.

5.6. Combining the Fragments

`combineFragments()` is passed an array of message fragments that may be missing a few pieces. Nevertheless, there's still value in saving what has been retrieved, and so a single string is constructed, with any null fragments labeled by error substrings. When `combineFragments()` returns, `reveal()` writes the string out to a text file.

```

private static String combineFragments(String[] msgFrag,
                                      int numFragFound)
// combine all the message fragments into a single string
{
    if (numFragFound == NUM_FRAGS)
        System.out.println("ALL message fragments extracted");
    else
        System.out.println("Only found " + numFragFound + "/" +
                            NUM_FRAGS + " message fragments");

    StringBuffer sb = new StringBuffer();
    for (int i=0; i < NUM_FRAGS; i++) {
        if (msgFrag[i] != null)
            sb.append(msgFrag[i]);
        else
            sb.append("\n????? missing fragment " + i + " ??????");
            // used when there's no fragment
    }
    return sb.toString();
} // end of combineFragments()

```

5.7. Evaluating the FragMultiStegCrypt Class

The principle reason for moving to multiple fragments in `FragMultiStegCrypt` was to make the stego message more resilient to image cropping. If you recall, `MultiStegCrypt` cannot handle images that are cropped along the x-axis (see Figure 13).

If the same cropping tests are performed using `FragMultiStegCrypt`, the message is retrievable in both cases. Even when the image is more severely cropped, as in Figure 14, only one fragment out of 100 is lost, and the rest of the text is perfectly readable.



Figure 14. Cropped Water Lilies along Both Axes (One Fragment Lost).

When ImageJ's image subtraction is applied to an image modified using FragMultiStegCrypt, the surface plot looks like Figure 15.

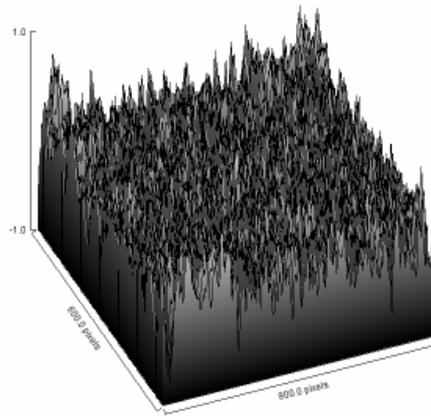


Figure 15. A Surface Plot of the Subtracted FragMultiStegCrypt Image.

Although it's very difficult to see, there seems to be a slight repeating pattern in the plot, which is perhaps due to the "XXX" header appearing at the beginning of all the fragments.

An issue with this approach is the large space overhead for each fragment. The header, fragment number, password, and message size fields add 12 bytes to each fragment, which requires 96 (12*8) bytes of storage in the image. This is quite excessive if the original message is small, and is split up into 100 parts of just 5-10 bytes (40-80 bytes in the image). One answer would be to rewrite the fragment generation code, to enforce a minimum size for a fragment.

FragMultiStegCrypt, is quite good at dealing with cropping, but it can't survive other common image operations, such as saving the image in a different format, blurring, or rotation. In those cases, the hidden message is lost.

There are a few ways that commercial steganography tools deal with these kinds of attack. One is to hide the message after changing the image using a Discrete Cosine Transform (DCT), which converts the pixel data into a collection of cosine functions of different frequencies. Hiding the data by modifying these functions makes it harder to delete the message with pixel-level operations, such as blurring.

Another approach is to hide the message in a more *visible* manner, so that it remains readable even after the image has been blurred or rotated. An example would be to add an extra flower to the water lilies picture, or perhaps change the shape of some of the foliage. Finding such changes can be difficult even when an eavesdropper has the

original image for comparison, as people who have tried "Spot the difference" puzzles will know. This technique can be viewed as a form of encryption since the intended reader must know the 'meaning' of the additional flower or changed leaves.