COMP8505 Assignment 2 Design Document

A00991905 Junyin Xia

Overview

To implement a “simple” least significant bit insertion (LSB) steganography tool.

Requirements

- Must support .bmp file extension
- Must implement LSB algorithm
- Must add encryption to payload data
- Must produce identical looking output to original image
- Must be able to retrieve embedded image
- Can use any language (JavaScript)

JavaScript File to Pixel Detailed Roadmap

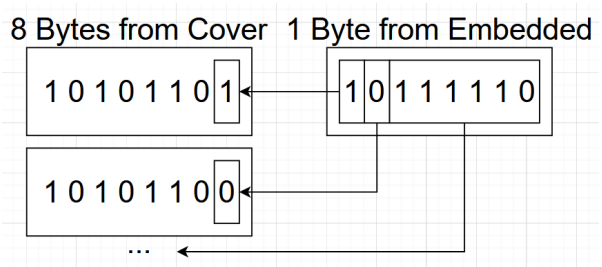
See js_img_roadmap.pdf

Application Full State Machine Diagram

See stego_encode_mode.pdf and stego_decode_mode.pdf

Issues to Solve

The concept of LSB steganograph is simple. You shave off the final bit from each byte of the cover file, and copy over a corresponding bit from the embedded file.



The first problem is that if we do this the file will become corrupted: not every byte from the cover file is available to us, only the pixel data section.

File Layout

The first challenge is to select a programming language that can take care of parsing out just the pixel data from a file. For example, this is the general file layout for a .bmp file (Wikipedia)

Structure name	Optional	Size	Purpose	Comments
Bitmap file header	No	14 bytes	To store general information about the bitmap image file	Not needed after the file is loaded in memory
DIB header	No	Fixed-size (7 different versions exist)	To store detailed information about the bitmap image and define the pixel format	Immediately follows the Bitmap file header
Extra bit masks	Yes	3 or 4 DWORDs[6] (12 or 16 bytes)	To define the pixel format	Present only in case the DIB header is the `BITMAPINFOHEADER` and the Compression Method member is set to either BI_BITFIELDS or BI_ALPHABITFIELDS
Color table	Semi-optional	Variable size	To define colors used by the bitmap image data (Pixel array)	Mandatory for color depths ≤ 8 bits
Gap1	Yes	Variable size	Structure alignment	An artifact of the File offset to Pixel array in the Bitmap file header
Pixel array	No	Variable size	To define the actual values of the pixels	The pixel format is defined by the DIB header or Extra bit masks. Each row in the Pixel array is padded to a multiple of 4 bytes in size
Gap2	Yes	Variable size	Structure alignment	An artifact of the ICC profile data offset field in the DIB header
ICC color profile	Yes	Variable size	To define the color profile for color management	Can also contain a path to an external file containing the color profile. When loaded in memory as "non-packed DIB", it is located between the color table and Gap1.[7]

We can only modify bytes from the Pixel array section, which will be in a slightly different place in every bmp file. For other file formats this will be different too: all this need to be taken care of by the image library. For this task, we choose **vanilla JavaScript** as its canvas class can parse pixel data out from all image file formats into a consistent uniform format.

Copying Bits

It turns out you can't copy bits between bytes the same way as items in 2 arrays, the bitwise operators (quite ubiquitous in most languages) don't allow that. In JavaScript, pixel data is stored as an array of unsigned 8bit integers. JS converts these integers to 32bit when doing bitwise operations.

```
255 << 1 (shift 255 left by 1 bit):
(255): 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111
(<<1): 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1111 1110
```

To copy the 8th bit from byte 129 from the embedded to the cover's byte 255, we prepare 255 like so first shifting right then left to bump the final bit off (Extra 0 bits on far left omitted here)

```
255       :  1111 1111
255 >>>1 :  0111 1111 ←  move right 1 space
255 << 1 :0 1111 1110    move left 1 space
```

Then we extract the 8th bit (index 7, bit value 1) from 129 by bumping and masking every other bit off

```
129   :           1000 0001
  <<4 : 0100 0000 1000 0000 move left 7 spaces
  &255: 0↓00 0000 1~~111~~ ~~1111~~ AND it together with 255 (all 1's in 8 last bits)
```

```
  >>>7: 0000 0001 0000 000 move right 7 spaces
```
Then we add it together, and set it back in the pixel data's array.
```
254 :1111 1110 + 0000 0001
255 :1111 1111
```

The Alpha Channel

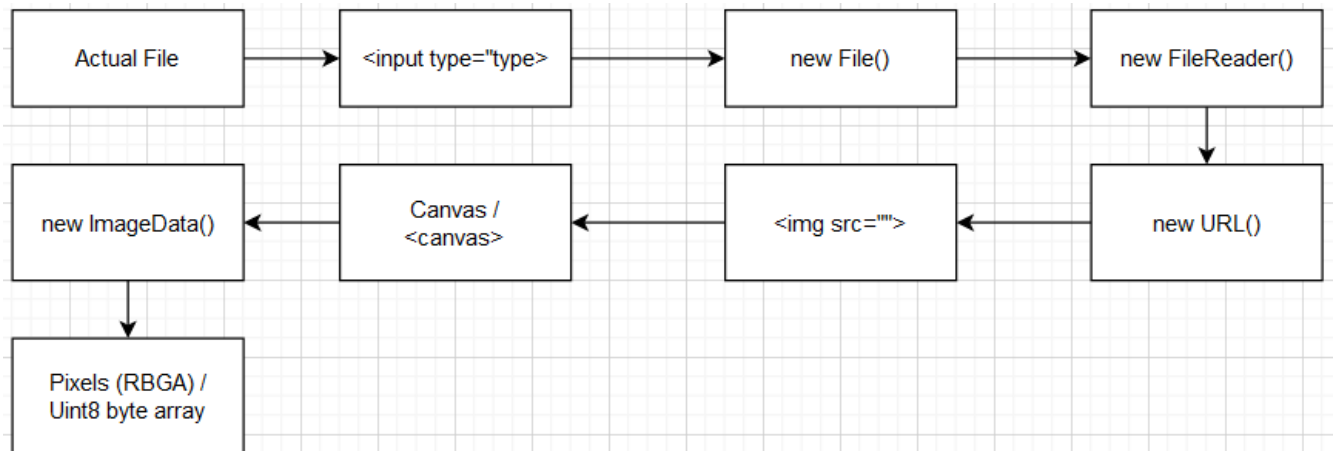
After choosing a language (JS) and playing around with the image library, I noticed some bits keep getting lost. Turns out this is a known issue with browser optimization. JS doesn't use RGB (3bytes per pixel), it uses RGBA (4bytes per pixel). The 4th byte, the alpha value, sets the pixel's opacity. If alpha is anything but 0 or 255, JS will multiply the other 3 bytes with the alpha value in a process called "premultiplied alpha",

eg Pixel (64, 128, 0, 128), A=128, 50% opacity → 0.5 * RBG = Pixel (32, 64, 0, 128)

This is great for loading images in the browser, but any lost bits make LSB steganography impossible. The workaround is to use the alpha channel to save data (awful, only 1/4 of cover can be used to carry payload), set alpha to 0 (bad, 0% opacity leaves transparent pixels in the final image, easy to notice something's off), or to set alpha to 255 (100%). I chose the last option: hardcoding all our pixels' alpha value to 255 will ignore the effects of browser optimization, keeping all our RGB bits unchanged, However we can only use 3/4 of our cover's payload capacity. In addition, png images loses their alpha chanel, which means our tool can only handle non-transparent images.

URLs

This is a simplified roadmap of how JS turns most uploaded image files into byte level pixel data



As you can see, the file will have to become a data URL at one of the steps. Working off of this, LSB insertion specifies how bytes must be embedded, but not **what** bytes are embedded. The **what** leaves some room for creativity.

I had 3 ideas for how to encode the embedded image into the cover's pixel RGB values

1. Encode the embedded's pixels' RBG values. My version 1 ended up with huge payload sizes. Not efficient at all.

2. Encode the embedded file's URL (a base64 copy of the file data in url format). This was my second version. It's not as efficient as storing the file for smaller to medium sized files. But urls make storing large files very efficient. So efficient, it was possible to store an image inside of itself using data Urls. Using Urls I can also very easily check if the password used is correct (by checking if the decrypted file is a valid url).
3. Encode the embedded file's bytes. The JS FileReader class actually has another option to parse the file as a binary byte array instead an URL. Through testing, I've found this is ends up with the smallest payload, very efficient. But encoding issues made this idea impossible. In the end I couldn't find out the proper encoding to put the embedded file bytes back together as a readable file.

In the end, I went with idea 2. JaveScript **really** likes when things are converted to URLs, and it was also the simplest method programming wise, as only the cover image needs to be turned from URL to pixels. Everything else stays an URL.

Encryption

Open-source JavaScript encryption libraries are surprisingly inefficient for this task. They are very secure, but add a ton of extra data to a payload. The Stanford Javascript Crypto Library for example turns a 1 char message into 150 chars. Not only is that overkill for this tool, LSB insertion is already very inefficient so I don't want to store extra data in my payload. So I threw together my own password-cipher encryption "dcutils::simpleEncrypt". It's based on a Caesar cipher with some reverse string and XOR logic (see below) thrown in cause I wanted an algorithm that handles both encryption and decryption so I didn't need to write separate functions. It's not that secure, but it does the job.

```
(string.reverse).reverse = string   (number XOR anything) XOR anything = number
```

File Layout

This is the final file layout I decided on, to be encoded bit by bit (literally) into the cover.

# of Bytes	Type of data	Notes
4	watermark	The string "STEG" to mark files encoded by this application.
2	2 digit int	Length of filename string (99 max)
X	Filename string	Filename of length X, where X <= 99
2	2 digit int	Number of digits in payload size
Y	Payload size Z	Payload size of Y digits, where Y <= 99, measured in chars/bytes
Z	Payload	Payload of size Z. This is a encrypted copy of the URL which will be reconstructed back into the image on the other side

Pseudocode

Encode logic

```
String coverImgPath =  args.get
String embeddedImgPath = args.get
Image coverImg = fopen(coverImgPath, "rb")
Image embeddedImg = fopen(embeddedImgPath, "rb")
if (coverImg == null || embeddedImg == null) {
      print("Bad files")
      exit
}
if (embeddedImgPath > 99) {
      print("Embedded filename too long! Max 99 allowed")
      exit
}
String password = args.get
String payload = encrypt(embeddedImg.readBytes(), password)
if (coverImg.filesize < payload.size * 8) {
      print("Cover too small!")
      exit
}
try {
      String filename = embeddedImgPath
      byte[] encodedByteArray = new byte[]
      String watermark = "STEG"
      LsbEncode(encodedByteArray, 4, watermark)
      LsbEncode(encodedByteArray, 2, filename.length.pad(2, ' '))
      LsbEncode(encodedByteArray, filename.length, filename)
      LsbEncode(encodedByteArray, 2, payload.length.pad(2, ' '))
      LsbEncode(encodedByteArray, payload.length, payload)
      String encodedImgUrl = convert encodedByteArray to url newImgUrl
      doc.downloadlink.href = newImgUrl
} catch (Exception encodeError) {
      print(encodeError)
      exit
}
```

Decode logic

```
String stegoImgPath = args.get
Image stegoImg = fopen(embeddedImgPath, "rb")
if (stegoImg == null) {
      print("Bad files")
      exit
}
String watermark = "STEG"
String chunk = LsbDecode(embeddedImg.readBytes(), 4))
if (chunk != watermark) {
      print("No hidden img here")
      exit
}
int X = Integer.parseInt(LsbDecode(embeddedImg.start().readBytes(), 2)))
String filename = LsbDecode(embeddedImg.readBytes(), X)
int Y = Integer.parseInt(LsbDecode(embeddedImg.fseek().readBytes(), 2)))
int payloadSize = Integer.parseInt(LsbDecode(embeddedImg.fseek().readBytes(), Y)))
String encryptedUrl = Integer.parseInt(LsbDecode(embeddedImg.fseek().readBytes(),
payloadSize)))
String payload = decrypt(encryptedUrl, password)
if (!isValidUrl(payload)) {
      print("Bad password")
      exit
}
doc.downloadlink.href = payload
```