Steganography with images. Sample Pairs and RS analysis resistance.

Miguel Canton Cortes
mcanton@ucdavis.edu

Amrit Subramanian
asubramanian@ucdavis.edu

Hossein Khosravani
hkhosravani@ucdavis.edu

March 20, 2014

Keywords: Steganography, Steganalysis

1 Introduction

A popular digital steganography technique is Least Significant Bit (LSB) embedding. With the LSB embedding technique the secrete message, possibly encrypted, is embedded in the LSBs of the pixels in a given image. Since this barely changes the original color, embedding message bits in the least significant bit plane will not cause any visually discernible difference from the original image. However, using statistical analysis it is quite easy to detect the tampering as this changes doesn't resemble ordinary noise. In this paper we are going to briefly describe two common steganalysis techniques used against LSB steganography and how we have developed a steganography program to defeat them.

2 Threat Model

Our threat model begins with the assumption that a third party has access to our modified image but doesn't know if it hides something nor has access to the original image. This modified image may be intercepted by monitoring the communication between the sender and receiver or may be publicly shared in a web page. As we are using the LSB embedding technique, we assume the attacker will test it against LSB detection tools which is our worst case scenario. Our objective is to keep our modification undetectable even when explicitly looked for using staganalysis techniques.

As a side note, we've implemented additional measures so that it would be virtually impossible for the attacker to recover anything meaningful even if he/she thinks the image is suspicious.

3 Steganalysis techniques

3.1 RS Analysis

RS steganalaysis is a method for detecting LSB embedding, which relies on dual statistics derived from spatial correlations in images. The image is divided into separate disjoint groups of pixels. The noise between adjacent pixels is then measured for each group using a special function known as a discrimination function. A mask is then applied to the groups which flips the LSBs of a certain group of pixels to determine whether the groups are regular,singular, or unchanged. A regular group is a group where the noise between pixels increased as a result of the LSB flipping, a singular group is a group where the noise decreased, and an unchanged group contains the same amount of noise. The inverse of the flipping function is applied to the groups and the numbers are compared. A normal image should have more regular groups than singular groups because flipping the LSB naturally induces noise and the group numbers before and after flipping the bits should be the same, which can be accounted for by applying the inverse of the flipping function to the counted regular and singular groups. These spatial correlations hold true for an unaltered image and the errors between these groups are analyzed to detect any anomalies. [1] [2]

Our approach in the project defeats this strategy by using compression and filtering the image before choosing which pixels to alter. The RS analysis strategy also gives rise to the theory that if a message of length p is embedded in a stego image, then p/2 of a stego image's pixels would be flipped. It then examines the regular and singular groups by looking at p/2 pixels and flipping them. By using compression, we defeat this effort by greatly decreasing the size of our encrypted message, thus even if the compressed message size was known, the spatial data would not match the amount of data actually embedded. Using a **Sobel filter** on the image defeats the noise measuring of RS analysis. The Sobel filter seeks out areas of high contrast in the image such as borders where colors are changing, and slightly modifies those colors. Thus the noise generated from embedding is hard to detect due to the drastic change in pixel colors in the original image itself.

3.2 Sample Pair Analysis

Sample Pair Analysis (SPA) is an approach to detecting least significant bit steganography in digital signals such as images and audio. This technique is based on a finite state machine whose states are selected multisets of sample pairs called trace multisets. Some of the trace multisets are equal in their expected cardinalities if the sample pairs are drawn from a digitized continuous signal. SPA method can be utilized easily, and its speed is fast, so it can be able to analysis a large number of images. Random LSB flipping causes transitions between these trace multisets with given probabilities and consequently alters the statistical relations between the cardinalities of trace multisets. By analyzing these relations and modeling them with the finite state machine, we arrive at a simple quadratic function that can estimate the length of embedded message with high precision. [3]

However an adversary can try to fool the detection method by avoiding hiding message bits at locations where some of adjacent sample pairs have close values. This is achieved

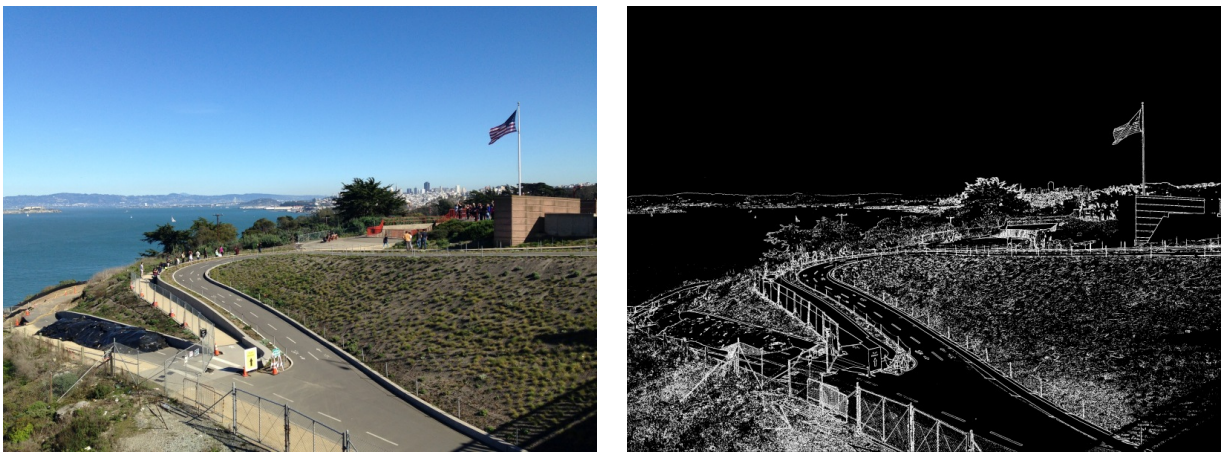
by detecting the image's areas with high contrast due to borders, for instance, using the aforementioned Sobel filter.

4 Hiding the message

If the message is sequentially embedded in the image it is easily detected as the noise in that part of the image would be distributed quite differently than in the rest of the image.

Our first approach is to randomly scatter the message through the image as this not only distribute the new noise more evenly but it also makes it much difficult to recover the message without knowing the sequence of pixels. This certainly makes less obvious the embedding and give good results when tested against detection tools as it can be seen in the last section but it still can be greatly improved being more aware of the pixels we are modifying.

Using a Sobel filter we can detect areas of high contrast in the image due to irregular surfaces or textures, borders or just the normal noise of a photo. If we only change this kind of pixels, a steganalysis wouldn't see anything odd due to the context of these pixels in the image.



(a) Original picture.

(b) Sobel output.

Figure 1: The Sobel filter with threshold outputs in white the pixels with high contrast in their surroundings and in black the pixels in smooth areas.

5 Program flow

In Figure 2 we can see an overview of the program flow. There are three main components:

1. We compress the file before embedding it in the image so it creates a smaller footprint. An encryption step is added to protect the message content in the unlikely event it is recovered by a third party.

2. We apply a border detection filter (Sobel filter) to get a list of pixels which can be modified with minimum chances to get detected.

3. Using the password's hash, we seed two Pseudo-Random Number Generators. One of which will shuffle the pixels sequence and the other will tell which RGB component we must use to insert the message's bit. The seeds used are 16 bytes long each. This random step makes virtually impossible recover the bit sequence in the correct order without knowing the seeds.

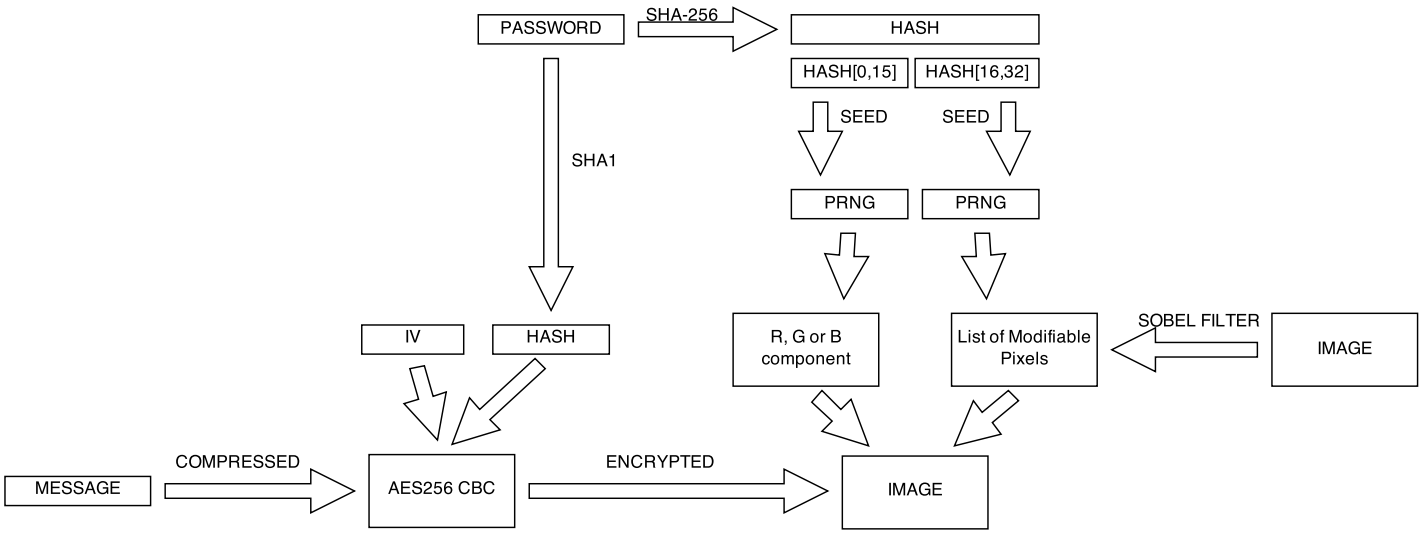


Figure 2

The program usage and compilation is described in Appendix A.

6 Results

We have tested three different variations of our algorithm, from the most basic to the most advanced: sequentially embedding the message, randomly embedding the message and randomly embedding the message using a pixels set obtained by applying the Sobel filter to the image. Each one have been tested against RS and Sample Pairs analysis and their output is the average percentage across all groups/colours.

6.1 Test 1

The file embedded for testing has a size of 11341 bytes. The image used is 1920 x 1200 pixels. Using the Random + Sobel technique, there's only a 2.73% increase compared to the original image bias when using the RS analysis and a 2.48% when using the Sample Pairs analysis. This would be completely unsuspicious as the natural bias of an image is usually bigger than that.

Steganalysis	Original image	Sequential	Random	Random + Sobel
RS	1.71995	1.88594	2.05257	1.76699
Sample Pairs	1.67196	1.83591	1.98447	1.71359

(a) Original picture. (b) Sobel. (c) Program output.

Figure 3: Image used for test 1. 1920 x 1200 pixels.

6.2 Test 2

The file embedded for testing has a size of 2189 bytes. The image used is 640 x 960 pixels.

Steganalysis	Original image	Sequential	Random	Random + Sobel
RS	2.14895	2.4905	2.49079	2.28913
Sample Pairs	1.79327	2.13156	2.1318	1.89807

(a) Original picture. (b) Sobel. (c) Program output.

Figure 4: Image used for test 2. 640 x 960 pixels.

7 Conclusions

The program we have developed successfully hides a message in a image without being detected by the most common staganalysis techniques with the assumption that the attacker doesn't have the original image. Further development could be made to increase the embedding ratio when using the Random+Sobel technique, perhaps using another filter as well.

References

[1] Jessica Fridrich, Miroslav Goljan: *Practical Steganalysis of Digital Images - State of the Art* (2002). Available at: ws2.binghamton.edu/fridrich/Research/steganalysis01.pdf

[2] Kaustubh Choudhary: *Properties of Images in LSB Plane*, IOSR Journal of Computer Engineering (IOSRJCE), Volume 3, Issue 5 (2012). Available at: www.iosrjournals.org/iosr-jce/papers/Vol3-issue5/B0350816.pdf

[3] Sorina Dumitrescu, Xiaolin Wu, Zhe Wang: *Detection of LSB Steganography via Sample Pair Analysis*, IEEE Transactions on Signal Processing (July 2003).

A Compiling and using the program

In order to compile the program, two libraries must be installed in the system:

- FreeImage (libfreeimage)
- OpenSSL (libcrypto)

A.1 Inserting

stegan -i [-s] inputImg messageFile password

The -s option activates the Sobel mode. "inputImg" is the original image. "messageFile" is the file to be embedded into the image. "password" is the key used to insert/extract.

The program will create a "cod_{inputImg}.png" image with the embedded message.

A.2 Extracting

stegan -e [-s] inputImg outputFile password

The "inputImg" argument must be an image previously created with the program using the same mode and the same password as given now as arguments.

B Source code

B.1 Makefile

```
1 stegan: main.cpp steganography.cpp
2   g++ -std=c++11 -O3 main.cpp steganography.cpp -lfreeimage -lcrypto -lm -o
        stegan
3
4 clean:
5   rm stegan
```

Listing 1: Makefile

B.2 main.cpp

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstring>
4 #include "steganography.h"
5
6 using namespace std;
7
```

```
 8 int main(int argc, char *argv[]) {
 9   Steganography steg;
10   ifstream in;
11   ofstream out;
12   unsigned char* buf = 0;
13   char filename[256];
14   size_t length;
15   bool wrongParameters = false;
16   bool sobel = false;
17
18   if(argc < 5)
19     wrongParameters = true;
20   else if(argv[1][0] != '-' || argv[1][2] != '\0' || (argv[1][1] != 'i' &&
       argv[1][1] != 'e'))
21     wrongParameters = true;
22   if(argc == 6) {
23     if(argv[2][0] == '-' && argv[2][1] == 's' && argv[2][2] == '\0')
24       sobel = true;
25     else
26       wrongParameters = true;
27   }
28
29   if(wrongParameters) {
30     cout << "Wrong parameters.\n";
31     cout << "Insertion:\tstegan -i [-s] inputImg messageFile password\n";
32     cout << "Extraction:\tstegan -e [-s] inputImg outputFile password\n";
33     exit(-1);
34   }
35
36   int offset = (argc == 6 ? 1 : 0);
37   if(argv[1][1] == 'i') {
38     in.open(argv[3+offset], ios::binary);
39     in.seekg(0, ios::end);
40     length = in.tellg();
41     in.seekg(0, ios::beg);
42     if(length == (size_t)-1) {
43       cout << "File '" << argv[3+offset] << "' not found\n";
44       return -1;
45     }
46     buf = new unsigned char[length];
47     in.read((char*)buf, length);
48     in.close();
49
50     if(sobel)
51       cout << "Inserting using Sobel mode\n";
52     else
53       cout << "Inserting using normal mode\n";
54
55     steg.loadImage(argv[2+offset]);
56     if(steg.embed((unsigned char*)argv[4+offset], strlen(argv[4+offset]), buf,
         length, sobel)) {
57       argv[2+offset][strlen(argv[2+offset])-4] = '\0';
58       snprintf(filename, 256, "cod_%s.png", argv[2+offset]);
59       steg.saveImage(filename, FIF_PNG);
60       cout << "File inserted correctly\n";
```

```
61      }
62      else {
63        if(buf)
64          delete [] buf;
65        return 1;
66      }
67    }
68    else if(argv[1][1] == 'e') {
69      steg.loadImage(argv[2+offset]);
70
71      if(sobel)
72        cout << "Extracting using Sobel mode\n";
73      else
74        cout << "Extracting using normal mode\n";
75
76      if(steg.extract((unsigned char*)argv[4+offset], strlen(argv[4+offset]), &
            buf, &length, sobel)) {
77        out.open(argv[3+offset], ios::binary);
78        out.write((char*)buf, length);
79        out.close();
80        cout << "File extracted correctly\n";
81      }
82      else {
83        cout << "Nothing extracted\n";
84        if(buf)
85          delete [] buf;
86        return 2;
87      }
88    }
89
90    if(buf)
91      delete [] buf;
92
93    return 0;
94  }
```

Listing 2: main.cpp

B.3 steganography.h

```
1  #ifndef _Steganography_H_
2  #define _Steganography_H_
3
4  #include <FreeImage.h>
5
6  class Steganography {
7    private:
8      FIBITMAP* image;
9      uint8_t* sobelImg;
10     unsigned height;
11     unsigned width;
12     size_t totalPixels;
13     struct Header {
```

```
14        size_t lengthUncompressed;
15        size_t lengthEncrypted;
16        uint8_t iv[32];
17      };
18      void getSHA256(const uint8_t* data, size_t length, uint8_t* hash);
19      void zlibCompress(uint8_t* in, size_t lengthIn, uint8_t** out, size_t*
            lengthOut);
20      void zlibUncompress(uint8_t* in, size_t lengthIn, uint8_t** out, size_t
            lengthOut);
21      void aesEncrypt(const uint8_t* data, size_t dataLen, uint8_t** encr,
            size_t* len,
22              const uint8_t* key_data, unsigned key_data_length, uint8_t* iv);
23      void aesDecrypt(const uint8_t* encr, size_t encrLen, uint8_t** data,
            size_t* dataLen,
24              const uint8_t* key_data, unsigned key_data_length, const uint8_t*
                    iv);
25      uint8_t applySobelFilter(unsigned x, unsigned y);
26      bool changesSobel(long x, long y, uint8_t rgb);
27      uint8_t readBit(unsigned x, unsigned y, uint8_t rgb);
28      void writeBit(unsigned x, unsigned y, uint8_t rgb, uint8_t bit);
29
30    public:
31      Steganography();
32      ~Steganography();
33      void loadImage(const char* file);
34      void saveImage(const char* file, FREE_IMAGE_FORMAT format, int flags = 0);
35      bool embed(const uint8_t* key, unsigned keyLength, uint8_t* data, size_t
            length, bool sobelMode = false);
36      bool extract(const uint8_t* key, unsigned keyLength, uint8_t** data,
            size_t* length, bool sobelMode = false);
37  };
38
39  #endif
```

Listing 3: Main class interface

B.4 steganography.cpp

```
1  #include <cstdlib>
2  #include <cstring>
3  #include <cmath>
4  #include <iostream>
5  #include <openssl/sha.h>
6  #include <openssl/aes.h>
7  #include <openssl/evp.h>
8  #include <random>
9  #include <algorithm>
10 #include <functional>
11 #include <iomanip>
12 #include <sstream>
13 #include <vector>
14 #include <unordered_set>
15
```

```
16 #include "steganography.h"
17
18 using namespace std;
19
20 Steganography::Steganography() {
21   width = height = 0;
22   totalPixels = 0;
23   image = 0;
24   sobelImg = 0;
25   FreeImage_Initialise();
26 }
27
28 Steganography::~Steganography() {
29   if(image)
30     FreeImage_Unload(image);
31   if(sobelImg)
32     delete [] sobelImg;
33   FreeImage_DeInitialise();
34 }
35
36 void Steganography::getSHA256(const uint8_t* data, size_t length, uint8_t*
       hash) {
37     SHA256_CTX sha256;
38
39     SHA256_Init(&sha256);
40   SHA256_Update(&sha256, data, length);
41     SHA256_Final(hash, &sha256);
42 }
43
44 void Steganography::zlibCompress(uint8_t* in, size_t lengthIn, uint8_t** out,
       size_t* lengthOut) {
45   size_t maxSize = (size_t)((double)lengthIn + (0.1 * (double)lengthIn) + 12);
46   *out = new uint8_t[maxSize];
47   *lengthOut = FreeImage_ZLibCompress(*out, maxSize, in, lengthIn);
48 }
49
50 void Steganography::zlibUncompress(uint8_t* in, size_t lengthIn, uint8_t** out
       , size_t lengthOut) {
51   *out = new uint8_t[lengthOut];
52   FreeImage_ZLibUncompress(*out, lengthOut, in, lengthIn);
53 }
54
55 void Steganography::aesEncrypt(const uint8_t* data, size_t dataLen, uint8_t**
       encr, size_t* len,
56                  const uint8_t* key_data, unsigned key_data_length, uint8_t*
                       iv) {
57   int written = 0;
58   size_t total = 0;
59   EVP_CIPHER_CTX e_ctx;
60   uint8_t key[32];
61
62   EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha1(), NULL, key_data,
         key_data_length, 5, key, iv);
63   EVP_CIPHER_CTX_init(&e_ctx);
64   EVP_EncryptInit_ex(&e_ctx, EVP_aes_256_cbc(), NULL, key, iv);
```

```
65
66    *encr = new uint8_t[dataLen + AES_BLOCK_SIZE];
67
68    for(size_t i = 0; i < dataLen; i += 2048) {
69      int toRead = (dataLen-i > 2048 ? 2048 : dataLen-i);
70      EVP_EncryptUpdate(&e_ctx, *encr+total, &written, &data[i], toRead);
71      total += written;
72    }
73    EVP_EncryptFinal_ex(&e_ctx, *encr+total, &written);
74    *len = total + written;
75  }
76
77
78  void Steganography::aesDecrypt(const uint8_t* encr, size_t encrLen, uint8_t**
        data, size_t* dataLen,
79                    const uint8_t* key_data, unsigned key_data_length, const
                        uint8_t* iv) {
80    int written = 0;
81    size_t total = 0;
82    EVP_CIPHER_CTX d_ctx;
83    uint8_t key[32], foo[32];
84
85    EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha1(), NULL, key_data,
          key_data_length, 5, key, foo);
86    EVP_CIPHER_CTX_init(&d_ctx);
87    EVP_DecryptInit_ex(&d_ctx, EVP_aes_256_cbc(), NULL, key, iv);
88
89    *data = new uint8_t[encrLen + AES_BLOCK_SIZE];
90
91    for(size_t i = 0; i < encrLen; i += 2048) {
92      int toRead = (encrLen-i > 2048 ? 2048 : encrLen-i);
93      EVP_DecryptUpdate(&d_ctx, *data+total, &written, &encr[i], toRead);
94      total += written;
95    }
96    EVP_DecryptFinal_ex(&d_ctx, *data+total, &written);
97    *dataLen = total + written;
98  }
99
100 uint8_t Steganography::applySobelFilter(unsigned x, unsigned y) {
101   const short dx[3][3] = {{1,0,-1}, {2,0,-2}, {1,0,-1}};
102   const short dy[3][3] = {{1,2,1}, {0,0,0}, {-1,-2,-1}};
103
104   if(x == 0 || x >= width-1 || y == 0 || y >= height-1)
105     return 0;
106
107   RGBQUAD color;
108   int sumX = 0, sumY = 0;
109   for(int m = -1; m <= 1; m++) {
110     for(int n = -1; n <= 1; n++) {
111       FreeImage_GetPixelColor(image, x+n, y+m, &color);
112       //int grey = 0.2126*color.rgbRed + 0.7152*color.rgbGreen + 0.0722*color.
              rgbBlue;
113       int grey = (color.rgbRed+color.rgbGreen+color.rgbBlue)/3;
114       sumX += grey * dx[m+1][n+1];
115       sumY += grey * dy[m+1][n+1];
```

```
116       }
117     }
118     //return sqrt(sumX*sumX + sumY*sumY) > 140 ? 255 : 0;
119     return abs(sumX)+abs(sumY) > 160 ? 255 : 0;
120 }
121
122 bool Steganography::changesSobel(long x, long y, uint8_t rgb) {
123     bool changed = false;
124
125     uint8_t oldBit = readBit(x, y, rgb);
126
127     writeBit(x, y, rgb, !oldBit);
128     for(long i = x-1; i <= x+1 && i < width && !changed; i++)
129       for(long j = y-1; j <= y+1 && j < height && !changed; j++)
130         if(i>=0 && j>=0 && applySobelFilter(i, j) != sobelImg[i+j*width])
131           changed = true;
132
133     writeBit(x, y, rgb, oldBit);
134
135     return changed;
136 }
137
138 uint8_t Steganography::readBit(unsigned x, unsigned y, uint8_t rgb) {
139   RGBQUAD color;
140     FreeImage_GetPixelColor(image, x, y, &color);
141     switch(rgb) {
142       case 0: return color.rgbRed & 0x01; break;
143       case 1: return color.rgbGreen & 0x01; break;
144       case 2: return color.rgbBlue & 0x01; break;
145       default: return 0;
146     }
147 }
148
149 void Steganography::writeBit(unsigned x, unsigned y, uint8_t rgb, uint8_t bit)
        {
150   RGBQUAD color;
151     FreeImage_GetPixelColor(image, x, y, &color);
152     switch(rgb) {
153       case 0:
154         if(bit) color.rgbRed |= 0x01;
155         else color.rgbRed &= 0xFE;
156         break;
157       case 1:
158         if(bit) color.rgbGreen |= 0x01;
159         else color.rgbGreen &= 0xFE;
160         break;
161       case 2:
162         if(bit) color.rgbBlue |= 0x01;
163         else color.rgbBlue &= 0xFE;
164         break;
165       default: break;
166     }
167     FreeImage_SetPixelColor(image, x, y, &color);
168 }
169
```

```
170 void Steganography::loadImage(const char* file) {
171   if(image)
172     FreeImage_Unload(image);
173   if(sobelImg)
174     delete [] sobelImg;
175
176   image = FreeImage_Load(FreeImage_GetFileType(file), file);
177   FreeImage_ConvertTo24Bits(image);
178   width = FreeImage_GetWidth(image);
179   height = FreeImage_GetHeight(image);
180   totalPixels = width*height;
181
182   FIBITMAP* bitmap = FreeImage_Allocate(width, height, 8);
183   sobelImg = new uint8_t[totalPixels];
184   for(unsigned x = 0; x < width; x++) {
185     for(unsigned y = 0; y < height; y++) {
186       uint8_t value = applySobelFilter(x, y);
187       sobelImg[x+y*width] = value;
188       FreeImage_SetPixelIndex(bitmap, x, y, &value);
189     }
190   }
191   FreeImage_Save(FIF_PNG, bitmap, "sobel.png");
192   FreeImage_Unload(bitmap);
193 }
194
195 void Steganography::saveImage(const char* file, FREE_IMAGE_FORMAT format, int
      flags) {
196   FreeImage_Save(format, image, file, flags);
197 }
198
199 bool Steganography::embed(const uint8_t* key, unsigned keyLength, uint8_t*
      data, size_t length, bool sobelMode) {
200   uint8_t hash[SHA256_DIGEST_LENGTH];
201   Header header;
202   uint8_t *compressedData=0, *encryptedData=0;
203   size_t lengthCompressed = 0;
204   vector<size_t> listCandidates;
205   unordered_set<size_t> chosen;
206
207   if(height == 0 || width == 0) {
208     cout << "Cannot insert: no image loaded\n";
209     return false;
210   }
211   if(length == 0) {
212     cout << "Cannot insert: no message\n";
213     return false;
214   }
215
216   getSHA256(key, keyLength, hash);
217
218   header.lengthUncompressed = length;
219   zlibCompress(data, length, &compressedData, &lengthCompressed);
220
221   aesEncrypt(compressedData, lengthCompressed, &encryptedData, &header.
        lengthEncrypted,
```

```
222           key, keyLength, header.iv);
223
224   delete [] compressedData;
225
226   listCandidates.reserve(totalPixels);
227   for(size_t i = 0; i < totalPixels; i++) {
228     if(sobelMode == false || sobelImg[i] == 255) {
229       listCandidates.push_back(i);
230     }
231   }
232
233   cout << "Message size:\n";
234   cout << "\tUncompressed:\t" << length << " bytes\n";
235   cout << "\tCompressed:\t" << lengthCompressed << " bytes\n";
236   cout << "\tEncrypted:\t" << header.lengthEncrypted << " bytes\n";
237   if(!sobelMode)
238     cout << "The maximum size for this image is: " << totalPixels/8-sizeof(
          Header)
239       << " bytes\n";
240
241   //Seed generators using hash(key)
242   seed_seq seed1(hash, hash+16);
243   seed_seq seed2(hash+16, hash+32);
244
245   shuffle(listCandidates.begin(), listCandidates.end(), std::mt19937(seed1));
246
247   std::uniform_int_distribution<int> distribution2(0,2);
248   std::mt19937 generator2(seed2);
249   auto diceRGB = std::bind(distribution2, generator2);
250
251   size_t position = 0, nBytesToWrite = sizeof(Header) + header.lengthEncrypted
        ;
252   uint8_t byte;
253   long x, y;
254   for(size_t i = 0; i < nBytesToWrite; i++) {
255     if(i < sizeof(Header))
256       byte = ((uint8_t*)&header)[i];
257     else
258       byte = encryptedData[i-sizeof(Header)];
259
260     for(unsigned bit = 0; bit < 8; bit++) {
261       uint8_t rgb = diceRGB();
262       bool valid;
263       do {
264         if(++position >= listCandidates.size()) { //No space
265           cout << "Cannot insert: file is too big\n";
266           delete [] encryptedData;
267           return false;
268         }
269         x = listCandidates[position]%width;
270         y = listCandidates[position]/width;
271         valid = true;
272         if(sobelMode) {
273           valid = !changesSobel(x, y, rgb);
274           if(valid) {
```

```
275                 for(long i = x-2; i <= x+2 && valid; i++)
276                   for(long j = y-2; j <= y+2 && valid; j++)
277                     if(i>=0 && j>=0 && i < width && j < height)
278                       if(chosen.find(i+j*width) != chosen.end())
279                         valid = false;
280               }
281               chosen.insert(x+y*width);
282             }
283           } while(valid == false);
284
285           writeBit(x, y, rgb, (byte>>bit)&0x01);
286         }
287       }
288
289       delete [] encryptedData;
290
291       return true;
292     }
293
294     bool Steganography::extract(const uint8_t* key, unsigned keyLength, uint8_t**
          data, size_t* length, bool sobelMode) {
295       uint8_t hash[SHA256_DIGEST_LENGTH];
296       Header header;
297       uint8_t *compressedData=0, *encryptedData=0;
298       size_t lengthCompressed = 0;
299       vector<size_t> listCandidates;
300       unordered_set<size_t> chosen;
301
302       if(height == 0 || width == 0) {
303         cout << "Cannot extract: no image loaded\n";
304         *data = 0;
305         *length = 0;
306         return false;
307       }
308
309       getSHA256(key, keyLength, hash);
310
311       listCandidates.reserve(totalPixels);
312       for(size_t i = 0; i < totalPixels; i++) {
313         if(sobelMode == false || sobelImg[i] == 255) {
314           listCandidates.push_back(i);
315         }
316       }
317
318       //Seed generators using hash(key)
319       seed_seq seed1(hash, hash+16);
320       seed_seq seed2(hash+16, hash+32);
321
322       shuffle(listCandidates.begin(), listCandidates.end(), std::mt19937(seed1));
323
324       std::uniform_int_distribution<int> distribution2(0,2);
325       std::mt19937 generator2(seed2);
326       auto diceRGB = std::bind(distribution2, generator2);
327
328       size_t position = 0, nBytesToRead = sizeof(Header);
```

```
329   uint8_t byte;
330   long x, y;
331   for(size_t i = 0; i < nBytesToRead; i++) {
332     for(unsigned bit = 0; bit < 8; bit++) {
333       uint8_t rgb = diceRGB();
334       bool valid;
335       do {
336         if(++position >= listCandidates.size()) { //No space
337           *data = 0;
338           *length = 0;
339           if(encryptedData)
340             delete [] encryptedData;
341           return false;
342         }
343         x = listCandidates[position]%width;
344         y = listCandidates[position]/width;
345         valid = true;
346         if(sobelMode) {
347           valid = !changesSobel(x, y, rgb);
348           if(valid) {
349             for(long i = x-2; i <= x+2 && valid; i++)
350               for(long j = y-2; j <= y+2 && valid; j++)
351                 if(i>=0 && j>=0 && i < width && j < height)
352                   if(chosen.find(i+j*width) != chosen.end())
353                     valid = false;
354           }
355           chosen.insert(x+y*width);
356         }
357       } while(valid == false);
358
359       if(readBit(x, y, rgb))
360         byte |= (1<<bit);
361       else
362         byte &= ~(1<<bit);
363     }
364     if(i < sizeof(Header)) {
365       ((uint8_t*)&header)[i] = byte;
366       if(i == sizeof(Header)-1) {
367         nBytesToRead = sizeof(Header) + header.lengthEncrypted;
368         if(nBytesToRead*8 > totalPixels) { // No sense
369           *data = 0;
370           *length = 0;
371           return false;
372         }
373         encryptedData = new uint8_t[header.lengthEncrypted];
374       }
375     }
376     else
377       encryptedData[i-sizeof(Header)] = byte;
378   }
379
380   aesDecrypt(encryptedData, header.lengthEncrypted, &compressedData, &
       lengthCompressed, key, keyLength, header.iv);
381   delete [] encryptedData;
382
```

```
383     zlibUncompress(compressedData, lengthCompressed, data, header.
            lengthUncompressed);
384     *length = header.lengthUncompressed;
385
386     delete [] compressedData;
387
388     return true;
389 }
```

Listing 4: Main class implementation