Graphical Interface for Watermarking

R. Amiot and M.A. Ambroze

Centre for Security, Communications and Network Research
Plymouth University, United Kingdom
e-mail: info@cscan.org

Abstract

An application was coded in C++, using Qt libraries, to run watermarking algorithms and attacks on images. Two algorithms were used: random LSB Insertion, and informed embedding. Four attacks were implemented: cropping, rotation, compression, blur. The output image after embedding can be compared to the cover work with the help of zooming and panning around.

Keywords

Watermarking, GUI, C++, Qt, trellis, informed embedding, convolutional code.

1 Introduction

Digital Watermarking is an important field for a variety of reasons; most of these have to do with protecting content producers. It is mainly applied to security and copyright (Cox & a, 2008). The field is very vast, allowing one, for example, to use the timing of packets over a network to encode information.

This application more specifically implements an algorithm modifying the pixel contents of a cover image, embedding an arbitrary message. The original image is not required to decode the message.

The design goals were to have an efficient interface, allowing to one select an algorithm and parameters, to visually witness the results of embedding on the image or to use various attacks on images.

This application was developed in C++, using Qt technology for the GUI, file IO, image edition and such. Qt was chosen for its versatility and portablity (tested on Windows XP SP3; and Mac Os X 10.5 and 10.6). The executable needs about 20Mb of .frameworks or .dlls to run.

2 UI Features

2.1 Operations on single images

The application can select an image from the user's hard drive with the help of a file dialog. The image is displayed on the left view port.

The user can set parameters, encode a message, or look for one in the image, using either a random LSB insertion algorithm or the Informed Embedding and Coding algorithm from J Cox & al (2004).

Should a message be encoded, the user can compare the output image and the input image: the output is displayed in the viewport to the right. It is possible to zoom and pan around on the images; they will be automatically synchronized. Finally, this window allows access to an important component, the Batch Operations window.

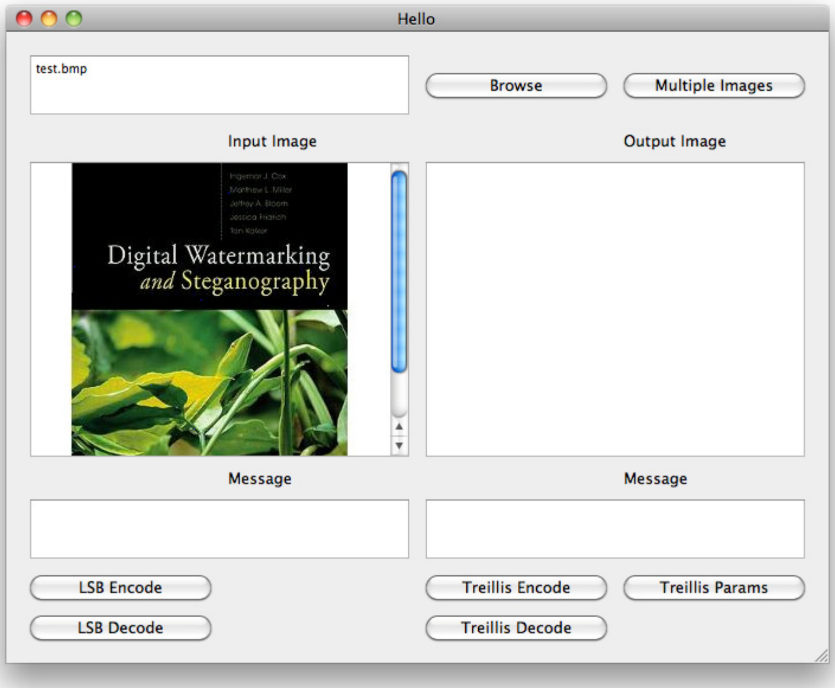
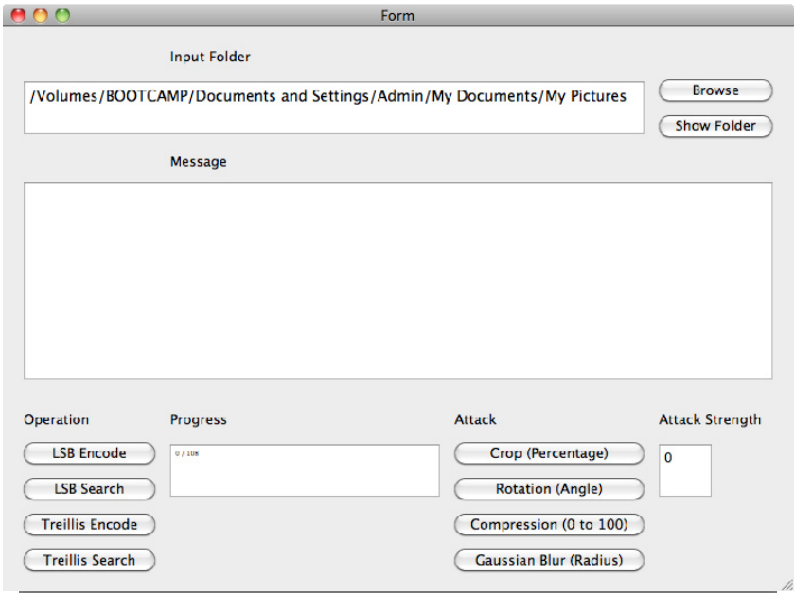


Figure 1: Interface for single images

2.2 Operations on image folders

The application can also work on folders, modifying all images within. It can either embed messages using any of the algorithms, search for a message (this returns the amount of images containing the image for LSB insertion, and the bit error rate for Informed Embedding), or apply an attack: cropping, rotation, compression, gaussian blur. The strength of the attack can be set by the user. There is no image viewport.



Input Folder: /Volumes/BOOTCAMP/Documents and Settings/Admin/My Documents/My Pictures

Progress: 0 / 108

Attack Strength: 0

Figure 2: Interface for whole folders

Given an attack parameter “x”, the image can be

- Cropped vertically and horizontally by x%
- Rotated by x degrees, around the center of the image
- Compressed with a strength x
- Blurred with a radius of x

Modified files, either by embedding or attacks, are saved in a specific subfolder, depending on the modification. The user is kept informed of the progress (eg, “x images done / y total images”).

2.3 Other Considerations

The user is informed of errors, insane parameters, and such by a popup covering the window. This popup closes whenever the user presses any key. File browsing behaviour is different when the operating system changes; it adapts to user habits there, and OS features. For instance, Mac users can get a preview of a selected image by pressing space. This is a very nice Qt feature. The program is CPU bottlenecked.

3 Algorithms

3.1 LSB Insertion

This algorithm randomly accesses pixels and modifies their parity in the blue channel's value according to the bits of the payload message. Decoding accesses the same bits but reads the parities and writes to message bits. This algorithm was modified to have visible effects (the blue channel values can go from 0 ↔ 255) to showcase the input/output comparison. Messages can be of any length, provided the image is large enough.

3.2 Informed Decoding

This algorithm follows what is described in Cox & al, 2004.

- The image's luminance is converted into a frequency representation via 8X8 block DCT.

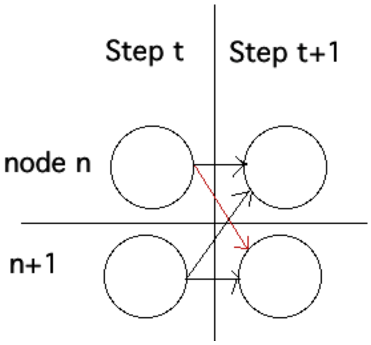


Figure 3: Arcs explanation

- A vector is extracted from the low frequency terms.
- A trellis is built according to user-defined parameters (number of arcs, number of nodes, random seed). This trellis is represented in memory by a set of arcs. Every node is set to have the same number of arcs exiting it than entering it. Arcs do not change over iterations. They have 3 properties: origin node, destination node, and a randomly generated label. In this figure, for the red arc, that would be n, n+1, and (float)rand(). Every arc also encodes either a 0 or a 1.

- A Viterbi decoder (Wesel, 2003) is run and returns the path (set of arcs) that is the most highly correlated with the vector extracted from the image. Correlation is done by comparing bits of the vector term at this step and the label of the arc.

- As each of these arcs encodes a bit, a message can be rebuilt from them.

3.3 Informed Encoding

This works the wau same as informed decoding, except that we only accept paths which encode the desired message. We then have an optimal vector, **g**, which we want to be the most highly correlated with the vector, despite noise.

3.4 Informed Embedding

We consider a "bad vector", **b**. This vector is the one decoded in the current state of the extracted vector, while adding random noise. The objective is to have **g** decoded instead of many possible **b**s, and the extracted vector is modified toward that objective. Should **g** be more highly correlated than 100 **b** by a certain amount (a user defined target) (every b is slightly different due to noise)**,** the extracted vector is considered acceptable, put back into the image, which is then converted back to pixels and saved. Unfortunately, this part of the algorithm doesn't work in that program: the original paper was not very clear on this step, and no matter the way of interpreting the relevant equations, the modified extracted vector grows less and less correlated with **g** with every update.

4 Conclusions

Despite setup difficulties on Os X, being rather CPU hungry, and some low level quirks, Qt proved ideal for this kind of application: it offered many powerful GUI features, and events, threads, many image operations, easy portability, easy UI layout design. Moreover, it made use of the host OS features.

Algorithm-wise, the coder could not draw results since the final embedding part was not done in time.

Should anyone want to finalize the application, here are some possible suggestions:

- Finalizing the trellis embedding algorithm
- Measuring duration of algorithmic calculations over a batch of images
- Implementing Plots
- Optimization: aside from some light parrallelization on Os X, performance was not considered at all
- Adding a scaling attack.

5 References

Cox, I. J., Miller, M. L., Bloom, J.A., Fridrich, J., Kalker, T., (2008), "Digital Watermarking and Steganography", second edition, Chapters 1 and 2 Morgan Kaufmann Publishers.

Cox, I.J., Miller, M.L., Doerr, G.J (2004), "Applying Informed Coding and Embedding to Design a Robust, High capacity Watermark", IEEE Trans. on Image Procesing, 13, 6, 792-807, June 2004.

Wesel, R. D. (2003). "Convolutional Codes". Encyclopedia of Telecommunications.