A GIMP Extension for Image Watermarking

Anandha Gopalan and Linian Liu
Department of Computer Science
Wichita State University
Wichita, KS 67260-0083
U.S.A.

Abstract

With the emergence of the World Wide Web and also the popularity of digital media, *digital watermarking* has become very important for copyright protection and authentication. Many watermarking schemes for images have been proposed, and they are evaluated according to four criteria : transparency, robustness, efficiency and payload. There is a need to compare these schemes quantitatively. In this project, we have implemented a software that provides a variety of benchmarking metrics that measure the *transparency* of embedded watermarks in images.

Keywords : Digital Watermarking, Benchmarking, GIMP.

1 Introduction

Digital watermarks are information embedded in digital data (such as pictures, text, sound, program code, etc.) for the purpose of ownership verification or authentication. The recent emergence of the World Wide Web as a new medium has stimulated works on their use and implementation. The success of digital watermarks in protecting intellectual properties will promote wider use of the Internet as a safe venue for storing and sharing information. Digital watermarking for images is done by embedding the information in images in a robust manner (slightly modify some bytes) so as to be able to retrieve it later for authentication purposes.

The majority of watermarking schemes have been designed for images. These schemes are evaluated in terms of four *goodness* criteria : *transparency* (check how visually are the watermarked image and the original image different), *robustness* (check how good and stable is the watermark), *efficiency* (find out how easy is it to watermark) and *payload* (amount of useful information in a watermark) of the watermark. There is a need to compare the various watermarking schemes quantitatively.

In this project, we have implemented a software that calculates a variety of popular benchmarking metrics, which reflect the transparency of the watermark. We have chosen to implement this software as an extension of the GIMP (GNUs Image Manipulation Program). The GIMP is the best image manipulation program available for the Linux platform. It also has built into it lots of features that we would want to use like rotation, scaling, jittering, filtering etc. Also, the GIMP supports various graphic file formats (jpeg, png, pgm, giff, tiff etc.). GIMP displays each image in a *drawable* which consists of one or many *layers*. GIMP is extremely extensible and expandable. It has an advanced scripting interface in Script-Fu, a scheme dialect.

The software used in this project was GIMP version 1.1.8. This was the developers version. The stable version was version 1.0. The reason for choosing this version was that the plug-in needed for our project `plug-in-layers-import` was available for this version (it works with gimp 1.1.x), it was also available for the earlier version (gimp 1.0.x), but we needed some more additional PDB-procedures (`gimp-layer-get-linked`, `gimp-drawable-set-image`) to be able to utilise it.

The rest of this paper is organized as follows. Section 2 provides an introduction to digital watermarking and benchmarking. It also provides a subsection on the GIMP and a small tutorial on Script-Fu. There is another subsection that talks about how we can combine two images in a variety of ways. Section 3 deals with the implementation of various benchmarking scheme as well as the implementation of two popular watermarking techniques, namely Patchwork and the NEC algorithm. Section 4 wraps up the paper with the conclusion. Here we talk about the problems that we faced during the implementation and also give some ideas on how the implementations can be improved.

2 Preliminaries

2.1 Digital Watermarking

Digital watermarking is nothing but 'hiding' information in digital documents (DVDs, images, pictures, digital documents etc.). This is conceptually very different from something like cryptography which deals with 'concealing' information. This embedded information is used later for copyright protection and authentication [7]. The information that is hidden has to conform to four criteria : transparency, efficiency, robustness and payload.

Before we carry on, it would be good to introduce some terms that we would be using quite often in this paper. These terms have been agreed upon by people working in this field [1]. Cover Object : This is the object that is used to embed the information into. Embedded Information : The information that we need to embed. This hidden information is called a *watermark*. Stego Object : The resulting object after the hidden information has been embedded. Key : The key used to hide the information in the cover object, it is this key that we would use during the detection of the *watermark*. Embedding : The method used

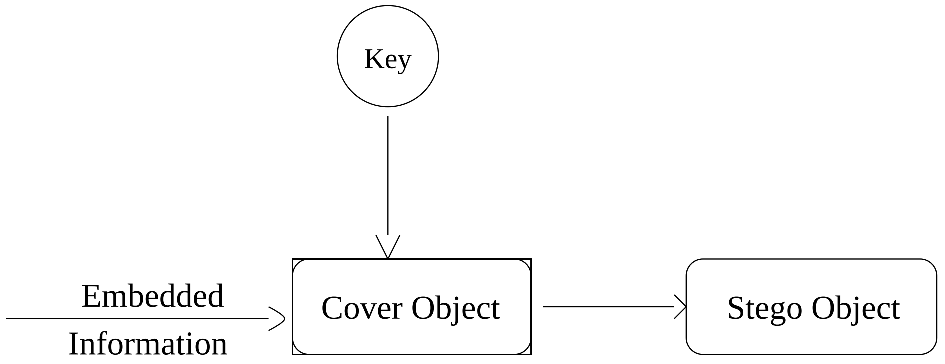


Figure 1: *Basic watermarking process*

for hiding information. Detection : The method of extracting the *watermark* from the stego object. Attacks : Methods used to try and manipulate or destroy *watermarks*.Attacker : Person who tries to destroy or manipulate the *watermark*.

A watermarking algorithm is complete only if we know both the embedding as well as the detection of the watermark. The reason we distinguish between attacks and attacker is very simple : attacks need not be caused by an attacker, for e.g : sending digital documents over the network could cause an appreciable amount of noise that can be classified as an attack, whereas an attacker is a person who knowingly wants to destroy the watermark.

2.2 Goodness Criteria

Until recently, digital watermarking did not receive much attention, but the growing rate of piracy and the ease with which digital media like DVDs, can be duplicated has been instrumental in getting a lot of people within the research community interested. This could have far reaching effects as it is envisioned that digital watermarking will solve a lot of issues like copyright protection, DVD access control, fingerprinting and advertisement. This is of utmost use for web designers, for eg : we watermark a picture and put it up on our site and it is copied and put up on another site, which claims ownership to the picture. The watermark in the picture would help resolve the issue if it is ever taken to court. Now that we know how useful watermarking can be, we should first examine what are the requirements of a good watermarking system.These are also the criteria of judging a 'good' watermarking system. These are collectively called the 'Goodness Criteria' :

2.2.1 Transparency

The *watermark* does not change the cover object appreciably (i.e), there is no apparent difference between the stego object and the cover object.

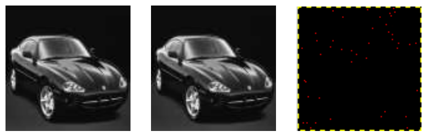


Figure 2: *Difference of the original and the patchworked image as a third image*

2.2.2 Robustness

Watermark cannot be removed without 'breaking' the cover object. What this means is that the watermark must be quite difficult to be removed and if one does succeed in removing it, then that process must have caused some degradation to the cover object as well. There is another kind of watermark in this category which is termed as *fragile.* These kinds of watermarks are destroyed easily upon manipulation to the cover object.

2.2.3 Efficiency

It must be easy to embed/detect a *watermark*, otherwise it defeats the purpose of watermarking if it is going to take a lot of time, in which case not too many people would be able to use it.

2.2.4 Payload

Watermark has to contain a reasonable amount of information, so that we can have some information to detect and make sure that we minimize the errors during detection. A 'good' payload is especially useful (i.e) in case the image is manipulated, then there is bound to some loss of data to the watermark, in case of a large payload, this would not be that significant. For a payload to be significant, it must be at least 70 bytes.

2.3 Benchmarking

Benchmarking requires a testbed on which to compare the different algorithms that are used for embedding information into images [6]. Most papers which highlighted various watermarking algorithms had their own series of tests, their own pictures and their own methodologies. So, there is no way in which we can compare the performance of any two algorithms without having to re-implement one of them. Re-implementing an algorithm might make the algorithm a lot weaker (or) stronger than it was earlier thus defeating our purpose of comparison. With a common testbed (or benchmark), one can have a rough idea as to how one algorithm works as opposed to another (in this case, all one needs is a common set of images that are watermarked with the respected algorithms). This also gives us the opportunity to find out how the algorithm is working and helps to correct the defects, if any to make the algorithm more robust. For benchmarking we compute the following visual distortion metrics [8] :

Average Difference

$$AD = \frac{1}{XY} \sum_{x,y} |p_{x,y} - p'_{x,y}|$$

Mean Square Error

$$MSE = \frac{1}{XY} \sum_{x,y} (p_{x,y} - p'_{x,y})^2$$

Signal-to-noise Ratio

$$PSNR = XY \max_{x,y} p^2_{x,y} / \sum_{x,y} (p_{x,y} - p'_{x,y})^2$$

Normalized Cross-correlation

$$NC = \sum_{x,y} p_{x,y} p'_{x,y} / \sum_{x,y} p^2_{x,y}$$

Correlation Quality

$$CQ = \sum_{x,y} p_{x,y} p'_{x,y} / \sum_{x,y} p_{x,y}$$

- where X and Y are the height and width of the images respectively. $p_{x,y}$ and $p'_{x,y}$ are the pixel values corresponding to the original and the watermarked image respectively.

2.4 Attacks

It is good that we have all the above metrics for benchmarking, but the basic problem with all of them is that they do not take into account the human vision system. So, an algorithm might appear robust and very good when compared to the benchmarking techniques, whereas the human eye could probably detect the difference between the original and the watermarked image. This is of course, not a desirable thing. Another problem that could happen in choosing the metrics is that it might be biased towards certain kind of algorithms, in the sense that certain types of algorithms that are based on a common technique would probably get a better rating when compared to the others.

Benchmarks are also subjected to attacks, this might seem a little strange, but when we say attacks, we mean it in a different sense. Suppose, we are benchmarking an algorithm and we get algorithm E as the one that has the best results and go ahead and implement it. A user might apply some of the following attacks to the image : low-pass filtering which is given by the following matrices :

Gaussian

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

Simple Sharpening

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

3x3 Median Filter

Each pixel is replaced by the median of the 3 x 3 neighborhood centered at that pixel.

Laplace

$$\begin{pmatrix} 1 & -2 & 1 \\ -2 & 1 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

The filtering is done as follows : we ignore the border pixels because they do not have all their nine neighbors. Starting from pixel (1,1) we do our calculations on every pixel till we reach pixel (w-1, h-1), where w is the width and h is the height. We take the matrix formed by the 3 x 3 neighborhood centered at that pixel and then calculate the sum of the product of the two matrices taken one element at a time. (i.e) we calculate

$$sum = \sum_{i=0}^{3} \sum_{j=0}^{3} nbhdmatrix[i, j] * filter[i, j]$$

For Laplace and simple sharpening cases, we replace the pixel value by *sum*. Since, we are working with gray scale images, if the value is less than 0, then we make it 0 or if the value is greater than 255 then we make it 255. In case of Gaussian filtering, *sum* is divided by the weighted mean which is the sum of all the elements in the filter matrix (=16). In case of the median filter each pixel is replaced by the median of the 3 x 3 neighborhood centered at that pixel. The median is calculated as follows : this is the middle element in a sorted array (descending or ascending). If the array has n elements, and if n is odd, then median is the (n+1)/2, element, if n is even, then the median is the mean of the n/2 element and the (n/2)+1 element.

The other types of attacks are JPEG compression, scaling, sropping, sotation and other simple geometric transformation. These would probably destroy the watermark that was embedded using algorithm E, and some other algorithm would have probably done better. What we are trying to say here is that the benchmarking technique is susceptible to geometric attacks. So, it might make sense to test for robustness on the basis of geometric attacks, but developing a testbed that takes both geometric as well as as noise attacks into account is much more difficult and research is going on in regard to this [6, 7].

2.5 Watermarking Algorithms

Without going into the historical aspects, we will now take a look at some modern day algorithms used to watermark images. Before we do this, we have to mention about a very important principle that is used. It is called Kerchoffs law (named after Auguste Kerchoffs, who stated this in 1883.), which states that : "The security of a watermarking scheme must be only in the choice of the key", what this basically means is that we cannot assume that the *attacker* has no idea of the scheme that we use.

2.5.1 LSB Algorithm

This was proposed by Tirkel and Osborne in 1993 [2, 3]. In this case the message is embedded in the least significant bit of a number of bits in a gray scale image (hence the name of the algorithm). There are two versions to this algorithm.

- Version 1 : Compress the 8-bit values for the color to 7-bit values, and embed information in the last bit. Detection is done by knowing the key which is basically the pseudo random number generator that we use to find the pixels to embed the information. Since the last bit is changed, there is always the danger of not meeting the transparency criteria, in the sense that we might end up changing the image to a large extent.

- Version 2 : Here the watermark is added to the least significant bit and not replaced as in the earlier case. This method is potentially better off as it does not change the image all that much. Extraction of the watermark is done by performing an auto correlation (this is a unique function and it gives us a unique value).

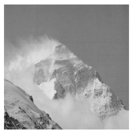


Figure 3: *Picture watermarked using LSB algorithm*

2.5.2 Patchwork

This algorithm works on the assumption that the given image has 255 levels of grayness, and that all brightness levels are equally likely [4]. Watermark insertion in the Patchwork watermarking system uses a secret key to seed a pseudo-random process that chooses pairs of pixels. For each pair of pixels, the brightness level of one of them is subtracted by a small constant, say k, while this is added to the brightness level of the other pixel. This is done a number of times (typically around 10000). Watermark extraction requires the same secret key used in insertion. The extraction process works by finding the same pairs of pixels that were chosen in the insertion process and analyzing the difference in contrast for each pair. This is done by taking the standard deviation of the original and comparing it to the standard deviation of the watermarked image. The reason for this is that by changing the pixel values of the image, we are actually changing the statistical dimensions of the image.

Figure 4: *Picture watermarked using patchwork algorithm*

2.5.3 NEC Algorithm

This algorithm was proposed by Cox, Killian, Lathan and Shamoon [5]. this algorithm takes the image and breaks it up into its Discrete Cosine Transform using the following formula :

$$DCT(i,j) = C(i)C(j) \sum_{x=0}^{N_1-1} \sum_{y=0}^{N_2-1} pixel(x,y) \cos\left[\frac{(2x+1)i\pi}{2N_1}\right] \cos\left[\frac{(2y+1)j\pi}{2N_2}\right],$$

where $C(i) = \frac{1}{\sqrt{N}}$ if i = 0, else $\sqrt{\frac{2}{N}}$ if $i > 0$.

Leaving the first discrete co efficient aside, we then use the first 1000 (this can vary) co efficients taken along the diagonal. For each C_i, we calculate $C_i = C_i + \alpha X_i$, where each X_i is a real number which is part of the pseudo-random number sequence (has normal distribution) and α is a real number whose value is normally 0.1.

The newly calculated co efficients are converted back into the pixel values using the following formula :

$$pixel(x, y) = \sum_{x=0}^{N_1-1} \sum_{y=0}^{N_2-1} C(i)C(j)DCT(i, j) \cos\left[\frac{(2x+1)i\pi}{2N_1}\right] \cos\left[\frac{(2y+1)j\pi}{2N_2}\right],$$

where $C(i) = \frac{1}{\sqrt{N}}$ if i = 0, else $\sqrt{\frac{2}{N}}$ if $i > 0$.

Detection is performed by again generating the sequence of real numbers and comparing the standard deviation of the original sequence to the sequence got from the watermarked image.

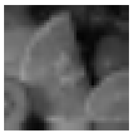


Figure 5: *Picture watermarked using NEC algorithm*

2.6 GIMP

2.6.1 Basics

The GIMP is an acronyn for GNU Image Manipulation Program written and developed under X11 for UNIX/Linux platforms. It is a freely distributed piece of software suitable for such tasks as photo retouching, image composition and image authoring. GIMP can be used as a simple paint program, a expert quality photo retouching program, an online batch processing system, a mass production image renderer, a image format converter, etc.

The GIMP is written by Peter Mattis and Spencer Kimball, and released under the GNU General Public License (GPL), version 2. The first version of GIMP version 0.54 was released in February 1996. Many other developers have contributed plug-ins and have provided support and testing. GIMP releases are currently being handled by Manish Singh. The newest version of the GIMP is version 1.1.19. The stable version is version 1.0. This version was released on June 5, 1998.

GIMP is extremely expandable and extensible. It is designed to be augmented with plug-ins and extensions to do just about anything. The advanced scripting interface allows everything from the simplest task to the most complex image manipulation procedures to be easily scripted. The basic functions of GIMP include Files And Preferences, Selections, Paint, Edit, Transform, Text, Brushes And Other Dialogs. GIMP also has full alpha channel support, plus sub-pixel sampling for all paint tools for high-quality anti-aliasing. GIMP supports many file formats, including gif, jpeg, png, xpm, tiff, tga, mpeg, pgm, pcx, bmp, and many others. GIMP can load, display, convert, save to many file formats.

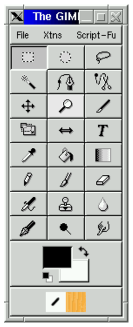
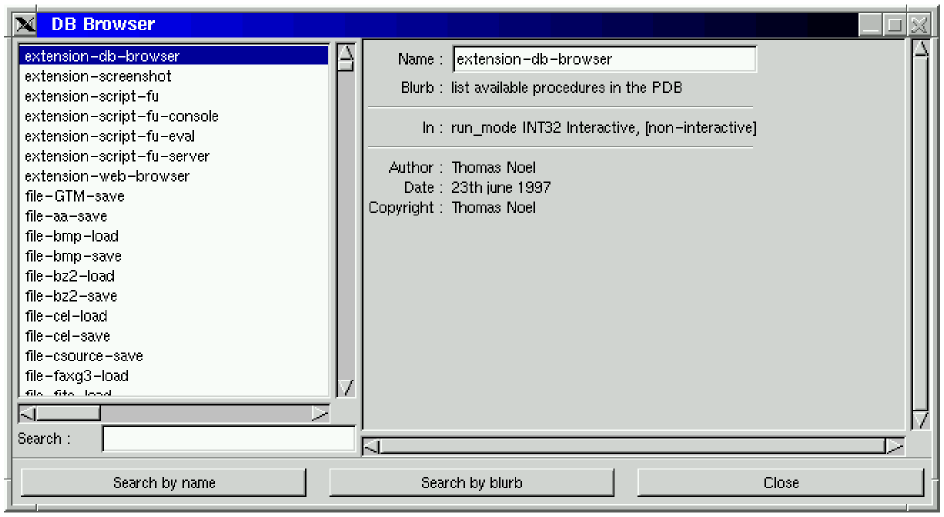


Figure 6: *GIMP user interface*

Script-Fu is the first GIMP scripting extension. Extensions are separate processes that communicate with the GIMP in the same way that plug-ins do. Plug-ins are external modules that actually do the nifty graphics transformations. The distinction is that extensions do not require an active image to operate on, instead extending the GIMP's functionality. In particular, the plug-in API has been made far more general with the advent of the procedural database (PDB).

The PDB allows the GIMP and its plug-ins to register procedures which can then be called from anywhere: internally, from extensions, and from plug-ins. There are already over 200 internal GIMP procedures, and more being created all the time. Because all of these procedures can be easily invoked from extensions, the logical next step was to create a scripting facility; thus, Script-Fu was born. Script-Fu is a macro language and based on Scheme, which is a interpreted lisp-like language. Script-Fu is a script-extension for the GIMP and connects to GIMP database.

GIMP Procedural Data Base (PDB) is correspondence between script functions and GIMP interface. What we use is SIOD. SIOD (Scheme in One Defun), is a small-footprint implementation of the Scheme programming language that is provided with some database, unix programming and cgi scripting extensions. An implementation such as SIOD usually runs by default in an immediate execution mode, where programs are parsed as they are entered, then executed with the results being printed [12].



extension-db-browser
extension-screenshot
extension-script-fu
extension-script-fu-console
extension-script-fu-eval
extension-script-fu-server
extension-web-browser
file-GTM-save
file-aa-save
file-bmp-load
file-bmp-save
file-bz2-load
file-bz2-save
file-cel-load
file-cel-save
file-csource-save
file-faxg3-load

Name : extension-db-browser
Blurb : list available procedures in the PDB
In : run_mode INT32 Interactive, [non-interactive]
Author : Thomas Noel
Date : 23th june 1997
Copyright : Thomas Noel

Figure 7: *Procedural Database Browser*

2.6.2 Architecture

If you are new to the world of image manipulation, then the concept of layers would seem strange, but you can think of layers as something lying on top of one another. This is how GIMP handles images. By default, each image has one layer to itself. We can always make duplicate layers, so at a time, we could have multiple layers on top of each other.

Layering allows for greater manipulation. Let us take an instance, suppose we want to paint something that has some drawing, some text, parts of some images, to make it simpler we could keep these elements in different layers, GIMP also gives us the option by which we can combine layers and also specify the combination mode. Even though we can have several layers on top of each other, we can have only one active layer to avoid confusion. GIMP makes sure that the layer is just the correct size, in case of images, the layer is the size of the image, in case of text, then the layer is the boundary of the text. This way a lot of space is saved.

The concept of a drawable is significantly different from that of layers. A drawable is a window which can have one or more layers. When an image is opened in GIMP, the image is given a layer and this layer is placed on top of a drawable. A drawable can have multiple layers, all lying on top of each other. We can see the various layers by right-clicking on the open image and clicking on the layers and channels option. The drawable is the working area for an image, of course one can increase the size of the drawable or decrease it.

2.6.3 Script-Fu Tutorial

Script-Fu uses the programming language scheme. Using Script-Fu, we can program in the environment provided by the GIMP. We can use most of the functions of the GIMP by calling the pre-defined procedures that exist in the GIMP database.

The scripts that we write must be stored in *$HOME/.gimp/scripts* directory. This is where GIMP checks for the presence of any scripts when it is loaded. The script files must have extension *.scm*. Once this is done, you need to refresh the GIMP, if GIMP is not already open, then open it and it will refresh itself and the script would appear, if GIMP is already open, then choose Xtns—Script-Fu—Refresh, this would make the script appear if there is no problems with it (like syntax errors or something like that). In this tutorial, the built-ins of Gimp are shown using this `font` and the scheme built-ins are shown using this font.

There are two kinds of scripts.

- Stand-alone : can always be used (button creation, etc.)
- Image dependent : dependent on image type (modifications)

Scheme is a programming language, a dialect of Lisp (List Processing) family of languages, generally utilizing a syntax based on parenthetical expressions delimited by whitespace, although alternative syntax capabilities are sometimes available [12]. Scheme processes LISTS. Each list is a function with its arguments. Scheme evaluates functions to return values. In scheme, each function is surrounded by (). Also, in scheme, all operators precede the operands, scheme uses the *prefix* notation. Any line after a ';' is ignored, so we can use it for comments.

(+ (* 3 4) (/ 4 2)) = 14

Variables

(set! orange '(250 240 0)) ;sets the value of orange

Declaring local variables

(let* ((a 3) (b 4) ...) (/ a b) ...)

The general form of a let* statement is: (let* (binding1 binding2 ...) expression1 expression2)

Functions

Functions in scheme have to be defined by using the define keyword. If these functions are to be used later they must be registered with the procedural data base (PDB). A function in Script-Fu is defined by :

(define (`script-fu-function` parameters1 parameter2 .....))

Every function in Script-Fu must be defined with the *script-fu* clause in front of it. This same name must be given while registering the function. The parameters are optional.

List Processing

Lists are made up of a first element (head) and the rest of the list (tail). The empty list () is also a list! car returns the head, cdr returns the tail, these two can be combined as shown.

```
(car '(1 2 3)) => 1
(cdr '(1 2 3)) => (2 3)
(cddr '(1 2 3 4)) => (3 4)
(caddr '(1 2 3 4)) => 3
```

Note : All GIMP functions return lists.

Image Manipulation

To find out the number and type of parameters for the built-in functions of GIMP that are mentioned here, do take a look at the procedural database (PDB). Before we work on images, we need to load them. GIMP provides a lot of native loaders to load the images, like `file-jpeg-load, file-gif-load`, a more generalized loader is `gimp-file-load` which loads the file regardless of the type of the file. This returns the id of the image that has been loaded, say this is img. After loading the image, we need to know the drawable of the image so that we can work on the image. This is done by the function `gimp-image-active-drawable`, this returns the id of the drawable, say draw. Now, that we have the drawable we can get the pixel values of the image at any given point x and y.

```
(set! pixel (gimp-drawable-get-pixel draw x y))
```

This returns a list which has as its head the number of channels and the pixel value of the point (x,y) as its tail. So, now that we have got this list we split it into two.

```
(set! bytes (car pixel)) ;Gets the number of channels.
(set! value (cadr pixel)) ;Gets the pixel values.
```

Once we have the pixel values (this is in the form of a one dimensional array), we can do some manipulations to it and then set the values back. The following section of code adds 1 to the red component of the pixel value. aref allows us to access that particular position of the array, eg : (aref arr 1) is equivalent to arr[1]. aset will set that particular array index value eg : (aset arr 1 10) is equivalent to arr[1] = 10.

```
(set! red (+ (aref value 0) 1)) ;Adds one to the red component
(aset value 0 red) ;Sets the value of the red component.
```

Now, that we have set the value, we need to set the pixel value of the drawable to the new value. This is done by

```
(gimp-drawable-set-pixel draw x y bytes (bytes-append value))
```

The reason we do not set the values directly is that the built-in `gimp-drawable-set-pixel` expects a byte array, and so we convert our array value into a byte array by using the scheme built-in `bytes-append`. Image is displayed using the `gimp-display-new` function, this takes the image id as its parameter.

Every Script-Fu function must be registered with the procedural data base (PDB). This way one can use our script in some other application just by calling the function and passing in the required parameters. We register our script by :

```
(script-fu-register
    "script-fu-function" ;Name of our function
    "<Toolbox>/Script-Fu/function" ;Where we want our function to appear
    "What the function does" ;Explain the nature of the function
    "Author" ;Author of this function
    "Copyright, 2000, Author" ;Copyright
    "Mar 5, 2000" ;Date the script was created
    ""
)
```

This tutorial has been designed to explain some of the concepts used in the implementation of the algorithms. A more comprehensive tutorial is available as part of the GIMP User Manual (part VII) [11]. A good tutorial on SIOD (Scheme in One Defun) is also available on the web [12].

2.7 Combining Images

Now, we will talk about a method that would allow us to visually find out the difference between two images, by combining two images (in other words, superimpose two images) using various combination modes. The fundamental question regarding implementing the various combination modes was how to superimpose images. The GUM (Gimp User Manual) indicated a way to do it, wherein you load the two images and then 'import' a layer from one image onto the other and then specify the combination mode. This feature was not available in the GIMP that we had (version 1.0). This was one of the reasons for choosing the developers version as the plug-in that we got called plug-in-layers-import was available for version 1.1. This plug-in allows one to import layers from one or more images into an images drawable (this is irrespective of the size of the images), after which we can specify the layer combination mode. This procedure is done in a series of steps.

2.7.1 Loading Images

First of all, we have to load the images that we want to work on, this is done by :

```
(fimg (car (gimp-file-load 1 filename filename)))
```

where fimg is the id of the image that has been loaded. Similarly, we can load the second image and call its id as simg.

2.7.2 Getting Drawable

To be able to import a layer from an image, one has to know the id of the drawable of the image. Here, we get the drawable id of the first image.

```
(fdraw (car (gimp-image-active-drawable fimg)))
```

where fdraw is the id of the drawable. We do not need the drawable of the second image as we are going to import the layer of the first image into the second image.

2.7.3 Import Layers

Now, we will import the layer of the first image into the second image using the plug-in plug-in-layers-import.

```
(plug-in-layers-import 1 simg fdraw fdraw 0 8)
```

where 1 is the run_mode which is non-interactive mode, 0 means that the layers gets copied/imported as is, 8 means that it copies/imports all the layers found in the drawable fdraw (this is useful in case an image has more than one layer).

2.7.4 Combining Layers

Now that we have imported a layer of an image into another image, we can specify the combination mode. Before that we must set the new layer as the active layer. This is done by :

```
(set! nlayer (car (gimp-image-get-active-layer simg)))
```

where nlayer is the id of the new layer which has been imported from the first drawable (fdraw) of the image.

We can now specify the combination mode of the layer.

```
(gimp-layer-set-mode nlayer mode)
```

where mode is the combination mode that we specify. The different combination modes available are :

0 - Normal	1 - Dissolve
2 - Behind	3 - Multiply
4 - Screen	5 - Overlay
6 - Difference	7 - Addition
8 - Subtract	9 - Darken only
10 - Lighten only	11 - Hue
12 - Saturation	13 - Color
14 - Value	15 - Divide

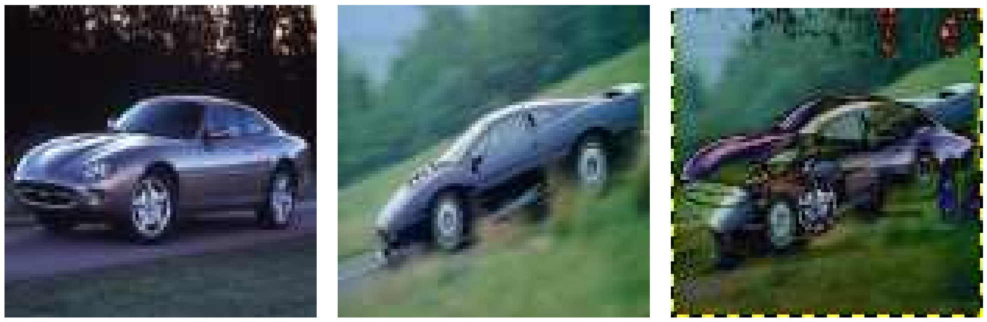
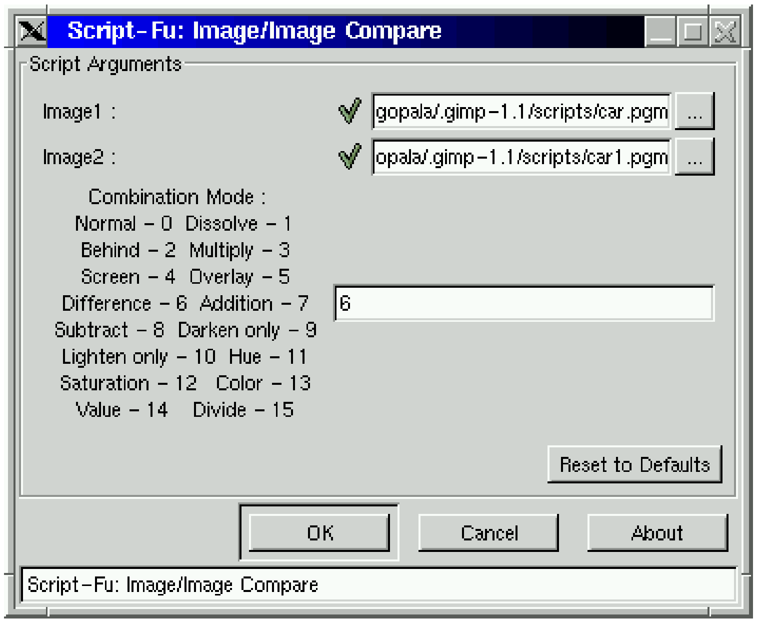


Figure 8: *Difference of the first two images as a third image*



Script-Fu: Image/Image Compare

Script Arguments

Image1 : gopala/.gimp-1.1/scripts/car.pgm

Image2 : opala/.gimp-1.1/scripts/car1.pgm

Combination Mode :
Normal - 0 Dissolve - 1
Behind - 2 Multiply - 3
Screen - 4 Overlay - 5
Difference - 6 Addition - 7
Subtract - 8 Darken only - 9
Lighten only - 10 Hue - 11
Saturation - 12 Color - 13
Value - 14 Divide - 15

6

Figure 9: *Image Compare Interface*

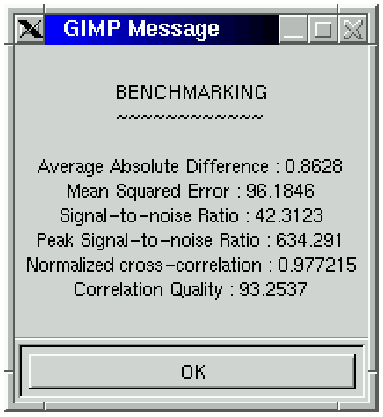
This is indeed very useful, as this gives us a way by which we can visually see the differences between two given images. The good thing about this is that (as mentioned earlier in section 2.6.3) it is independent of the type of image file. Also, this would work for images that are not of the same size. The plug-in used here checks if the second image is larger than the first, if so, it blindly imports the layer from the first image(this way the imported layer is smaller than the existing layer and so any changes to the mode is visible only in that part). If the first image is larger, then the layer that is imported is of the same size as the second image (in which case, the whole layer of the first image is not imported).

3 Implementation

3.1 Benchmarking

Implementing this using the GIMP was not very difficult. This is one of the reasons that we based our software on top of the GIMP, it already gives us the tools to load these image and get the pixel values of the images. This is done by a built in called `gimp-drawable-get-pixel`, this gets the pixel value of a point in the image, by iterating through the whole image one can get the pixel value at every point in the image. For loading the image, GIMP has this built in routine called `gimp-file-load` that called the appropriate file handler depending on the type of the image, so this makes our software image independent.

Being a image manipulation program, GIMP does have problems handling some of the large numbers that occur because of the benchmarking metrics. Also, the built in `gimp-drawable-get-pixel` is very slow as it has to call some basic built in routines.



```
BENCHMARKING
~~~~~~~~~~~~~

Average Absolute Difference : 0.8628
Mean Squared Error : 96.1846
Signal-to-noise Ratio : 42.3123
Peak Signal-to-noise Ratio : 634.291
Normalized cross-correlation : 0.977215
Correlation Quality : 93.2537
```

Figure 10: *Benchmarking of the image which was watermarked using the patchwork algorithm, length of the watermark was 1000*

3.2 Implementing Algorithms

Before we started to implement the algorithms, we used the GIMP to get used to the way one has to program in that environment. To program in GIMP, one has to program using Script-Fu which used scheme as its basic language (a small tutorial is given in section 2.6.3).

These algorithms were first implemented on small images (100 x 100) to make sure that they worked. Also, these images had a black background so that it would be clear to the human eye once the embedding was done. Initially, while using Patchwork and NEC, information was hidden in the blue channel and then extended to hide information in the pixel as a whole.

Design Principles

- User Interface : It provides convenient user interface. To achieve the fist purpose, we use SF-FILENAME parameter in Script-Fu-register function:

  `SF-FILENAME` ”Input file name” (`string-append` ”” gimp-data-dir ” /fish.jpg”)

  This will create a widget in the control dialog. The widget consists of a button containing the name of a file. If the button is pressed, a file selection dialog would pop-up.

- Image Types : It can deal with various image formats. The watermarks may be embedded not only in a gray scale image, but in RGB image as well. We just change the blue channel of an RGB image to satisfy the ‘transparency’ requirement for a good watermarking system.

- Seeding : It can produce a unique seed. We choose the current time as the key and use it to reset the algorithm seed for rand function to produce pseudo-random numbers.

3.2.1 Patchwork

Embedding

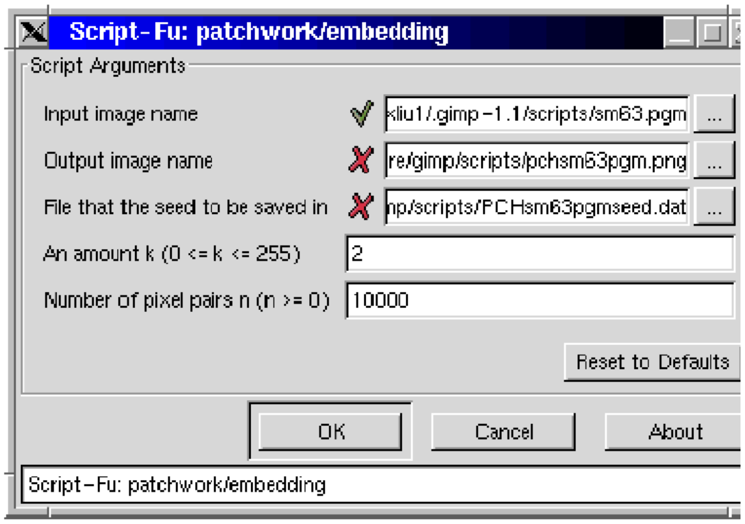
First we get the current time as the seed for rand function. The seed is stored in a default seed file ”PCHseed.dat” in the current directory (but user can specify his/her own seed file). This is important, because the detector needs the same seed during decoding. This is done by

(`set!` seed (realtime)) ; Get current time (`srand` seed) ; Reset seed

Secondly, we use a pseudo-random number generator to choose pixel pair (An, Bn) from the original image.

Thirdly, we modify the brightness level of An and Bn by an amount k:

(`aset` pixelarray index (+ (`aref` pixelarray index) k))) and then put them back to the output image. Repeat this for n steps (n typically around 1000). At last, display the original image and the watermarked one.


Script-Fu: patchwork/embedding

Script Arguments

Input image name ✓ kliu1/.gimp-1.1/scripts/sm63.pgm

Output image name ✗ re/gimp/scripts/pchsm63pgm.png

File that the seed to be saved in ✗ np/scripts/PCHsm63pgmseed.dat

An amount k (0 <= k <= 255) 2

Number of pixel pairs n (n >= 0) 10000

Figure 11: *Patchwork Interface*

Detection

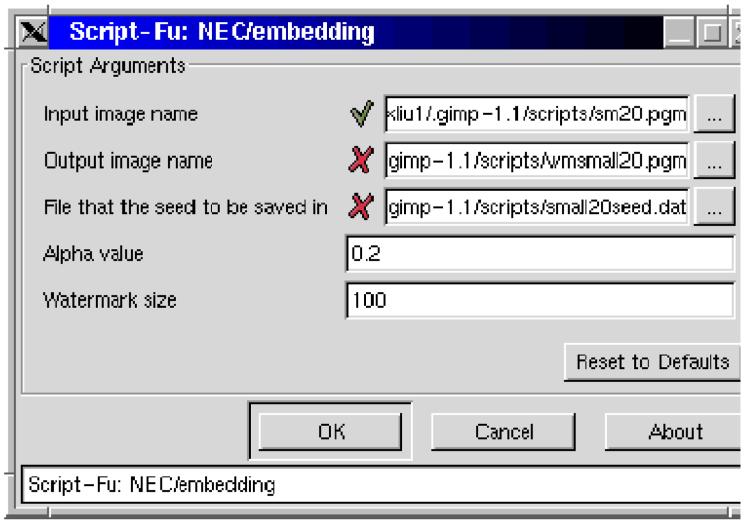
We open the seed file to get the key so that we can run the same pseudo-random number generator to find the same pairs of pixels chosen during the embedding algorithm. Then, we, compute the sum of the difference for each pair, standard deviations SD, and sum/SD. After these calculations, we print out the final result.

3.2.2 NEC Algorithm

Data Structure

- In a spatial domain, we do not need to create a two dimension array to store pixels of an image, because through GIMP we can get a pixel value directly from an image. This can save time and space.

- In a transform domain, we use a list instead of a two dimension array to store DCT array. This is because that SIOD (Scheme in One Defun) can only allocate one dimension array currently. The length of the DCT list is the number of rows of the original image. Each element in the list is also a list with the length of the number of columns of the image. That is, each element stores one row of the image.



Input image name ✓ xliu1/.gimp-1.1/scripts/sm20.pgm

Output image name ✗ gimp-1.1/scripts/wmsmall20.pgm

File that the seed to be saved in ✗ gimp-1.1/scripts/small20seed.dat

Alpha value 0.2

Watermark size 100

Figure 12: *Nec Interface*

- The watermarks are random numbers with distribution of N(0,1). They are stored in an array with length of the size of watermark specified by the user.
- We define our own pseudo-random number generator gaussrand instead of simple function rand. The gaussrand function can generate random numbers with distribution of N(0,1).

Embedding

The following steps are followed in the embedding process.

- The image to be watermark is loaded and the seed is initialized using the current time. The seed is stored in a file so that we can retrieve it during detection. Before the image is loaded, the user is asked for the name of the image, the watermark size and the alpha value.
- The image is broken down into pixels and is converted into its Discrete Cosine Transform (DCT).
- The random numbers are generated (say, N) and the first N DCT co effcients (excluding the first) are chosen along the diagonal and their values are modified.
- The modified DCT array is once again converted into its corresponding pixel values and these new pixel values are put back into the image so that the image is now *watermarked*

Detection

The following steps are followed in the detection process.

- The original image and the watermarked image are loaded. The seed that was stored in a file earlier is read and the original pseudo random numbers are generated again (this is the original watermark). The DCT co efficients of both the original and the watermarked image are calculated.
- Using the DCT coefficients of the original and the watermarked image and the alpha value the watermark is extracted from the second image.
- The original watermark and the extracted watermark are taken and the standard deviation is calculated. If this is over the given threshold then we can safely say that the image has been watermarked.

4 Conclusion

In this paper we have talked about the benchmarking techniques and also of a technique which allows us to combine the original and watermarked images into one and allows us to do some interesting manipulations. We have also succeeded in integrating the two. Both the benchmarking and the image combination were done using two different functions written using Script-Fu, all we had to do was to write a third function and call the functions written earlier. This was made possible because every Script-Fu function that is written in GIMP is registered with the Procedural Database (called PDB, one can see all the functions by invoking the DB browser). In this way, we have two ways to view the embedded watermark, one is by using the mathematical technique provided by the benchmarking and the other is by viewing the transparency of the watermark by using the image combination software.

At the beginning of this paper, we had mentioned some of the problems regarding using GIMP to do the benchmarking. One of the major problems was that GIMP was not able to handle big numbers very well. To cite an example : To make sure that the benchmarking techniques had been implemented correctly, we took the same image and ran the algorithm on it. If the image size was below 60x60, then we got the expected values (AD = 0, MSE = 0, SNR = 0, etc), if the size exceeded this (or roundabout) then the values were not as before. The reason for this, as we concluded was that after a point of time, the numbers become too large for GIMP to handle and it ends up generating some garbage values. This is not a problem otherwise, as in the long run this evens out (if we consider the garbage as an error, then in the end all the errors kind of cancel each other out).

For people who are not familiar with scheme, it may seem a daunting task to program in the GIMP environment initially. There is another plug-in available which allows us to write scripts using Perl. We did try to down load that plug-in and install it but were not successful as we did not have all the libraries required by it. You can visit the web site for more information [10]. Depending on the version you have you can download the correct plug-in.

In the benchmarking field, there is still a lot of research going on as people are trying to establish the correct testbed that would be 'fair' to all methods in its rating. The basic problem is that the current method is susceptible to attacks (not exactly intended in the same way as regards with watermarks) and does not provide a fair benchmark regarding geometric transformations done to watermarked images. The problem is that the technique does not take into account the type of the attack that the watermark has been under, it just calculates the same metric regardless of the attack.

We need to take into account the type of the attack, as otherwise we would not know which technique would work the best for us depending on the media used (audio, images, video etc). For example, if we are benchmarking an audio watermarking scheme, then it is of no use trying to take geometric attacks into consideration [6].

References

[1] B. Pfitzmann Information Hiding Terminology *Information Hiding, Springer Lecture Notes in Computer Science*, v 1174, pp. 347–350, 1996.

[2] R. G. van Schyndel, A. Z. Tirkel, C. F. Osborne. A Digital Watermark. *Proceedings of the 1994 1st IEEE International Conference on Image Processing*, Part 2 (of 3). Austin, TX. pp. 86–90.

[3] R. G. van Schyndel, A. Z. Tirkel, C. F. Osborne, W. J. Ho, N. R. A. Mee and G. A. Rankin. Electronic Watermark. *Digital Image Computing, Technology and Applications (DICTA'93)*, pp. 666–673, Macquarie University, Sydney, 1993.

[4] W. Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *I. B. M. Systems Journal*, vol. 35, no. 3&4, pp. 313–336, 1996.

[5] I. J. Cox, J. Kilian, T. Leighton and T. Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE transactions on image processing*, vol. 6, no 12, pp. 1673–1687, 1997.

[6] M. Kutter and F. Petitcolas. A fair benchmark for image watermarking systems. *In Ping Wah Wong and Edward J. Delp, editors, proceedings of Electronic Imaging '99, Security and Watermarking of Multimedia Contents*, vol. 3657, pp. 226–239, San Jose, California, U.S.A., 25–27 Jan. 1999.

[7] F. Petitcolas, R. J. Anderson and M. G. Kuhn. Information hiding – A survey. *Proceedings of the IEEE, special issue on protection of multimedia content* 87(7):1062–1078, July 1999.

[8] Stefan Katzenbeisser and Fabien Petitcolas (Editors). *Information Hiding Techniques for Steganography and Digital Watermarking*. Artech House Books, 1999.

[9] The GIMP web site is at : http://www.gimp.org.

[10] The GIMP plug-in registry is available at : http://registry.gimp.org.

[11] The GIMP User Manual written by, Karin Kylander and Olaf S. Kylander is available at : http://manual.gimp.org/manual/GUM/GUM.html.

[12] The SIOD (Scheme in One Defun) tutorial is available at : http://people.delphi.com/gjc/siod.html.