Implementation of Invisible Digital Watermarking on Image Nonlinearly of Arithmetically Compressed Data

Sabyasachi Samanta
Haldia Institute of Technology
Haldia, WB, INDIA

Saurabh Dutta
Dr. B. C. Roy Engineering College Durgapur,
WB, INDIA

Abstract
This paper presents how long textual information are encrypted and compressed to a number of floating point numbers and it to binary equivalent; then to substitute that stream of bits some suitable nonlinear pixel and bit positions about the image depending upon the key; also to retrieve that hidden data bits from that pixels and to decompress into its original information.

Key words:
pixel, invisible digital watermarking, arithmetic coding, nonlinear function.

1. Introduction

Digital Watermarking describes the way or technology by which anybody can hide information, for example a number or text, in digital media, such as images, video or audio. Arithmetic compression technique takes a stream of input symbols and replaces it with a single floating point number less than 1 and greater than or equal to 0. That single number can be uniquely decoded to exact the stream of symbols that went into its assembly**.** Most computers support floating point numbers of up to 80 bits or so. This means, it's better to finish encoding with 10 or 15 symbols. Also conversion of a floating point number to binary and binary to same floating point number is maximum time erroneous. Arithmetic coding is best to accomplish using standard 16 bit and 32 bit integer mathematics. A pixel with 32 bit color depth consists of α value, R (Red), G (Green) and B (Blue) value. α value is the value of opacity.

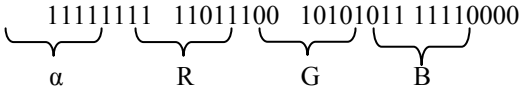

Figure 1.1 Bit Representation of a 32-bit RGB Pixel

First 8 bits of the 32 bits are reserved for this opacity (transparency of the image) value. If α is 00000000 the image is fully transparent. Each of three(R, G & B) 8-bit blocks can range from 00000000 to 11111111(0 to 255) [1] [2] [3] [4].

In this paper, we have proposed a technique, initially to encrypt through a table and then to encode as a real number in an interval greater than or equal to 0 and less than 1. First, we have assembled the table taking a number of characters or symbols available in keyboard or the special symbols as per user's prerequisite. Each characters (Ch) is assigned a range (r_c) indicated by high (H_c) and low (L_c) range between 0-1. Then taking a set of characters (maximum 10), a group (G) is defined. Each group is also assigned a range (r_g) indicated by high (H_g) and low (L_g) range. Hence, we can put maximum 100 characters with unique probability range in table. After that the long message is broken into a number of small messages. Every short message(less than or equal to 9 characters) is converted into two floating point numbers, one for character range (taking r_c) and the other one for where or in which group (G) the character belongs(i.e. taking r_g). Then we have transformed it to unsigned long integer (removing the floating point) and to equivalent binary number. Then we have replaced that stream of bits in nonlinear pixel and bit positions, in any one of last four significant bit of R,G & B at selected pixels about an image using nonlinear function and the private key cryptography technique taking the α value as 255 or as in the original image [5] [6].

Example: for a text with 24 characters, will be encrypted to an array of 168 *(8+2*((2*30) + (1*20)))* bits of stream. If we use ASCII-8 (American Standard Code for Information Interchange) to encode 192 bits are required. In an image with resolution of 800 X 600 has 2, 40,000 pixels. In our work only we are altering any one bit of last four significant bit of each R, G & B. Here maximum 56 pixels will be affected by this process. If any bit generated from characters become same to the targeted bit of image then there will be no change.

Section 2 represents the scheme followed in the encryption technique**.** Section 3 represents an implementation of the technique. Section 4 is an analytical discussion on the technique. Section 5 draws a conclusion.

Manuscript received April 5, 2010
Manuscript revised April 20, 2010

2. The Scheme

This section represents a description of the actual scheme used during implementing "Implementation of Invisible Digital Watermarking on Image Nonlinearly of Arithmetically Compressed Data" technique. Section 2.1 describes the encryption technique using four algorithms 2.1.1, 2.1.2, 2.1.3 & 2.1.4. While section 2.2 describes the decryption technique using two algorithms 2.2.1 & 2.2.2 [7].

2.1 Encryption of data bits about the image

2.1.1 Assignment of range for individual characters and groups

Step I: Take special characters/symbols or characters available in keyboard.
Step II: Count the number of characters (chlen) and calculate number of groups (nogp=chlen/10) and (extch=chlen%10).
Step III: Assigned range (r_c) to each characters/symbols indicated by high (H_c) and low (L_c) range between zero to one.
Step IV: Taking a set of characters (maximum 10 characters) define a group (G).Also assigned a range (r_g) indicated by high (H_g) and low (L_g) range to each of these.

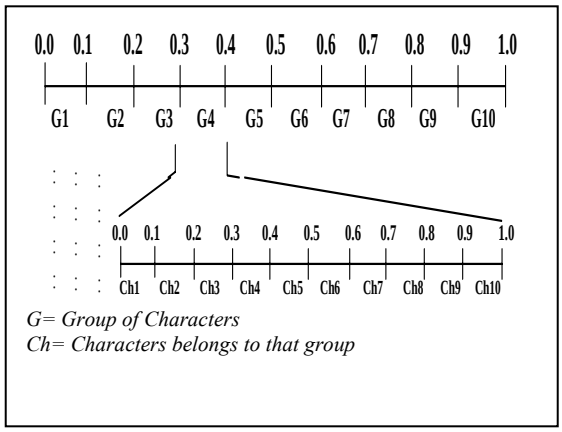

Figure 2.1.1.1 Subdivision of Encoded Characters

Step V: If extch =0 then repeat *Step II* to *Step IV* for i= *1* to *nogp*.
Otherwise repeat *Step II* to *Step IV* for i= 1 to (nogp+1).
Step VI: Stop.

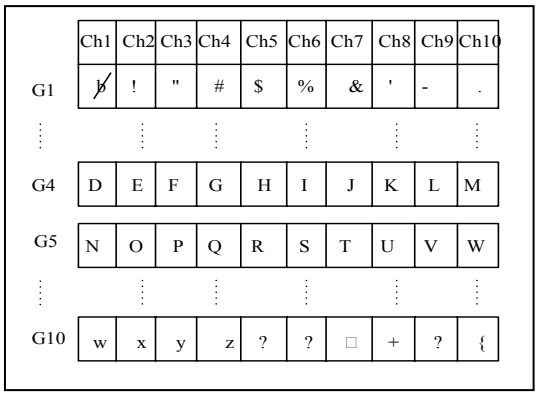

Figure 2.1.1.2 Characters in Group

Characters	Range for Characters ($L_c \le r_c < H_c$)	Range for Groups ($L_g \le r_g < H_g$)
#	$0.3 \le r_c < 0.4$	$0.0 \le r_g < 0.1$
&	$0.6 \le r_c < 0.7$	$0.0 \le r_g < 0.1$
A	$0.7 \le r_c < 0.8$	$0.2 \le r_g < 0.3$
B	$0.8 \le r_c < 0.9$	$0.2 \le r_g < 0.3$
E	$0.1 \le r_c < .0.2$	$0.3 \le r_g < 0.4$
I	$0.5 \le r_c < .0.6$	$0.3 \le r_g < 0.4$
L	$0.8 \le r_c < .0.9$	$0.3 \le r_g < 0.4$
N	$0.0 \le r_c < .0.1$	$0.4 \le r_g < 0.5$
O	$0.1 \le r_c < 0.2$	$0.4 \le r_g < 0.5$
R	$0.4 \le r_c < 0.5$	$0.4 \le r_g < 0.5$

Table 2.1.1.1: Characters and Groups with High and Low Ranges

2.1.2 Encode message using arithmetic coding and store it to encrypted array as binary values

Step I: Take characters as input from keyboard or special characters (which must be in Table 2.1.1.1).
Step II: Calculate the string length (chlen) from input.
Step III: Convert the length (chlen) into its 8-bit binary equivalent. Store that data bits to earr[bit] as LSB (Least Significant Bit) to earr[1] and MSB (Most Significant Bit) to earr[8] respectively.
Step IV: Calculate number of broken messages n = chlen/9 and remaining characters r =chlen%9.
Step V: Taking the range from Table 2.1.1.1 apply arithmetic coding technique to encode the character set into a single floating point number in between 0 - 1.
Set low to 0.0
Set high to 1.0
While there are still input characters do
get an input character

code range = high - low.
high = low + range*high_ range
low = low + range*low_ range
End of While

Continue this process to get input (1 to 9) for first n times and (1to r) at (n+1) th time and stop. Output the low.

Step VI: From the Low value and High value convert Low value (low) to an unsigned long integer number removing the floating point.

Step VII: Convert that number into equivalent binary number. For n times if total number of 0 or 1 is less than 30 then left fill with 0's. For the case of r:

a) If (r≥6 || r=0) as in Step VII.
b) If (r≥3 || r<6) and if total number of 0 or 1 is less than 20 then left fill with 0's.
c) If (r>0 || r<3) and if total number of 0 or 1 is less than 10 then left fill with 0's.

Step VIII: Store the binary values, LSB to earr[9] and rest to earr[bit] respectively.

Step IX: Repeat *Step V to Step VIII*

a) If r=0 then for i= 1 to (2*n) times
Otherwise for i= 1 to (2*(n+1)) times taking r_c and r_g from separately and respectively.

Step X: Stop.

2.1.3 Selection of the nonlinear pixel positions from the key

Step I: Take a four digit (decimal numbers from 0 to 9 except four successive 0's) key (***K***) as an input.

Step II: Take the value of *bit* from array e*arr[bit]* to calculate total number of pixels is required as three following data bit replaced in R, G & B of every pixel. So calculate number of pixel p= (ceil (bit /3)).

Step III: Take the key (K) and calculate the value of function

$$F(x, y) = K^p \text{ [i.e. POW (k, p)].}$$

Step IV: Store the exponential long double values into file one by one.

Step V: Repeat *Step III* to *Step IV* for i= (1 to p) and go to *Step VI.*

Step VI: Read the values as character up to "e" of the every line of the file and store it to another file with out taking the point [.].

Step VII: Modify the value as numeric and store it to an array *arrxyz[p]*

Step VIII: Take most three significant digit to *arrx [p],* next three digits to array *arry [p]* and last significant digit to *arrz[p]*.

Step IX: Repeat *Step VI* to *Step VIII* up to end of the file.

Step X: Stop.

2.1.4 Replacement of the array elements with R, G & B values of pixels

Step I: Calculate the width (w) and height (h) of the image.

Step II: Set x=*arrx[p]* and *y=arry[p]*.

Step III: To select the pixel position into image, compare the value of x and y with the value of w and h (where addressable pixel position is (0, 0) to (w-1, h-1)).

a) If (x >(w-1)) or (y >(h-1)) then
Set P (x, y) = P (0+(x %(w-1)), (0 +(y %(h-1)))
Otherwise Set P (x, y) = (x, y).

Step IV: To select the bit position (b) of selected pixel i.e. with which bit the array data will be replaced. Set *z =arrz [p]*.

i) If (z%4=0) then b=LSB
ii) If (z%4=1) then b=2nd LSB
iii) If (z%4=2) then b=3rd LSB
Otherwise b=4th LSB of each R, G & B of a pixel.

Step V: To replace the array elements with the selected bit position of selected pixel and to reform as a pixel

a) After reading the values of R, G & B convert each to its equivalent 8-bit binary values.
b) Replace subsequent element of earr[bit] by *following Step III to Step IV*.
c) Taking values of R, G & B switch it to the pixel value and place it to its position of the image (taking α value as before).

Step VI: For replacing the array element to pixels using the above mentioned process starting from the 0th element up to the end of the array.

A) If bit%3 = 0
Go to *Step VII.*
B) If bit%3 = 1

for 0th element to (bit-1)th element of the array repeat Step VI (A).For (bit)th element to R, value for G and B will be remain same. And go to S*tep VII.*

C) If bit%3=2

for 0th element to (bit-2)th element of the array repeat Step VI (A).For (bit-1)th element to R, (bit)th to G and B will be remain same. And go to S*tep VII.*

Step VII: Verify the pixel or bit positions which previously have used or not about the image.

a) If ((P(x, y)= (P(x, y)) || P (x, y)= P (x++, y++)) && (b=b++])then
Set P ((x, y), b) =P (0, h*)* and b as *Step IV*.
Repeat Step VII (a) for j=1 to p;
Repeat Step VII (a) for k=j to p.
Go to *Step VIII.*

Step VIII: Repeat *Step II to Step VII* for i=1 to p.

Step IX: Stop**.**

2.2 Decryption of the data bits from the image

2.2.1 Retrieving the replaced bits from the encrypted image

Step I: Take the same key (K) input as it was at the time of encryption.
Step II: To get the pixel positions with in the image and bit position in R, G & B of selected pixels run through *Step III* to *Step IX* of *Algorithm 2.1.3* and *Step I* to *Step VIII* of *Algorithm 2.1.4.*
Step III: Retrieving the encrypted bits from the selected bit positions of delectated pixels store it to decrypted array from darr[1] to darr[bit] respectively.
Step IV: To get the length repeat Step II to Step III for i= 1 to 3 times (as every pixel contain three data bits).
Step V: Taking data bits of darr [1] as LSB and darr [8] as MSB calculate the length (chlen) of message.
Step VI: To find out the total number of bits in decoding array (i.e. value of *bit* in *darr[bit]*), calculate n=chlen/9 and r=chlen%9 (as in *Step IV Algorithm 2.1.2).*
Step VI: Taking the value of n and r calculate bit= (2*(30*n + rbit) +8).
Where *rbit* are calculated as

a) If (r≥6) then *rbit =30*
b) If(r≥3 || r<6) then *rbit =20*
c) If(r>0 || r<3) then *rbit=10 (as in Step VII algorithm 2.1.2).*

Step VII: Now go through *Step III to Step IX of Algorithm 2.1.3* and *Step I to Step VIII of Algorithm 2.1.4 for i=4 to p.* Store the data's *darr[10] to darr[bit]* respectively.
Step VIII: Stop.

2.2.2 Decompress array elements to text using decoding algorithm

Step I: To translate the data bits to floating point number
a) If n≠0, assign the value *darr[8+n*i]* to LSB and *darr[8+30*n]* to MSB respectively and covert it to equivalent decimal number**.**
If the total number of digits of that decimal number is less than nine left filling with 0's translate it to nine digits floating point number**.**
b) If r≠0, assign the value *darr[(8+30*n) +1]* to LSB and da*rr[(8+30*n)+1)+rbit]* to MSB respectively and covert it to equivalent decimal number**.**
If the total number of digits of that decimal number is less than *r* left filling with 0's translate it to *r* digits floating point number**.**
Step II: Apply the decoding algorithm of arithmetic coding to convert it into individual range.
Get encoded number
Do
Find range from the table
Output the range
Subtract symbol low value from encoded number
Divide encoded number by range
Until no more symbols or zero.
Step III: Comparing the ranges of first time iteration of *Step II* and second time iteration of *Step II* alongside (i.e. comparing the range r_c and the r_g at same time) from the table (Table 2.1.1.1) find out the encoded characters (Ch).
Step IV: Executing the work of *Step (a)* go to *Step (b)*
a) If n≠0 repeat *Step I (a)* to *Step III* for i= 1 to 2 times. Repeat *Step IV for i=1* to *n* times.
b) If r≠0 repeat *Step I (b) to Step IV* for i=1 to 2 times. And Go to *Step VII.*
Step V: Finally put the characters one by one and assemble the original message**.**
Step VI: Stop.

3. An implementation

Let the message to be encrypt is NONLINEAR.
So the length of the message
=09(Decimal equivalent)
=00001001(8 Bit Binary equivalent)
All the characters of the message are defined in the Table 2.1.1.1 with their distinct range. Starting from the range 0.0 to 0.1 *Ch1's* (maximum10 characters from *G1* to *G10*) are defined. In that range the first character N is also defined. Applying the technique described in Algorithm 2.1.2 we get the codeword for the r_c as

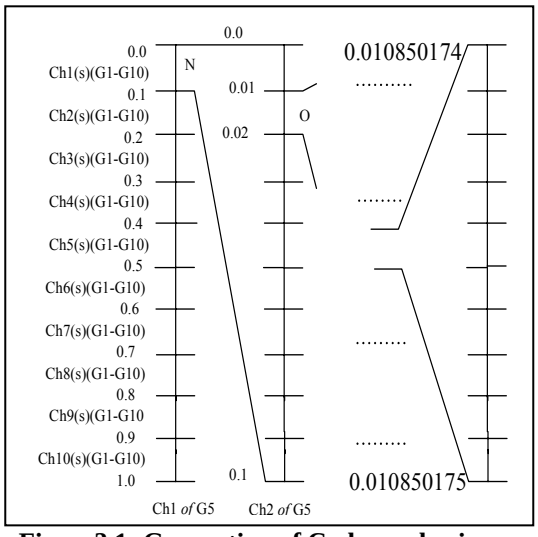

Figure3.1: Generation of Codeword using r_c

Hence we get the codeword for characters

0.010850174 ≤codeword_for_Ch
< 0.01050175.

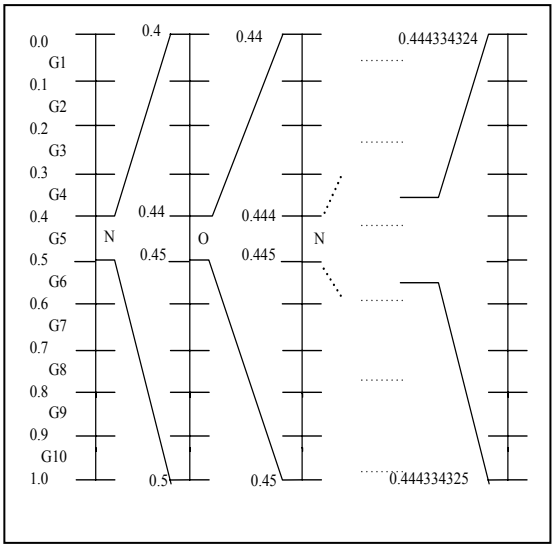

Figure 3.2: Generation of Codeword using r_g

And for groups we get
0.444334324≤codeword_for_group
<0.444334325
Taking the low values for characters
Codeword
=0.010850174(floating point number)
=10850174(integer number removing floating point)
=00000010100101100011110111111 10
(30 bit binary equivalent)
And taking low value for group we get
codeword=0.444334324 (floating point number)
=444334324 (integer number removing floating point)
=01101001111100000000011110100(30 bit binary equivalent)
First store the bits form length to encrypted array earr[bit], then store bits of stream from code words respectively as,

bits for length	*bits for character*	*bits for group*
earr[1]=1	earr[9]=0	earr[39]=0
: :	: :	: :
earr[8]=0	earr[38]=0	earr[68]=0

Figure 3.3: Data Bits in Encrypted Array

Let the key (K) =6379
The image size= 800 X 600(w x h)
Hence, number of affected pixel (p) = [ceil (68/3)] =23

In table 3.1, how the array data's are replaced with R, G & B values in selected nonlinear pixels of an image is described (following Algorithm 2.1.3 & Algorithm 2.1.4).

Key(K),i	Value	Value of pixel P(x,y)	Bit position b= Z%4	Array data to replace
6379,1	6.379000 e+03	P(637,300)	1st LSB	earr[1] earr[2] earr[3]
:	:	:	:	:
6379, 5	1.056241e+19	P(105,024)	2nd LSB	earr[13] earr[14] earr[15]
:	:	:	:	:
6379, 23	3.230789e+87	P(323,079)	1st LSB	earr[67] earr[68] B as same

Table 3.1: Replacement of Data Bits about Image

Thus we can transmit the encrypted watermarked image through any communication channel. Afterward applying the decryption technique as described in Algorithm 2.2.1 & Algorithm 2.2.2 we will be able to get back the encrypted message from that watermarked image in the decryption end. Taking character range and group range we get the encrypted characters which are defined in Table 3.2.

Codeword (Using r_c)	Codeword (Using r_g)	Character Range	Group Range	Character
0.010850174	0.444334325	$0.0 \leqslant r_c < 0.1$	$0.4 \leqslant r_g < 0.5$	N
0.10850174	0.44334325	$0.1 \leqslant r_c < 0.2$	$0.4 \leqslant r_g < 0.5$	O
0.0850174	0.4334325	$0.0 \leqslant r_c < 0.1$	$0.4 \leqslant r_g < 0.5$	N
0.850174	0.334325	$0.8 \leqslant r_c < 0.9$	$0.3 \leqslant r_g < 0.4$	L
0.50174	0.34325	$0.5 \leqslant r_c < 0.6$	$0.3 \leqslant r_g < 0.4$	I
0.0174	0.4325	$0.0 \leqslant r_c < 0.1$	$0.4 \leqslant r_g < 0.5$	N
0.174	0.325	$0.1 \leqslant r_c < 0.2$	$0.3 \leqslant r_g < 0.4$	E
0.74	0.25	$0.7 \leqslant r_c < 0.8$	$0.2 \leqslant r_g < 0.3$	A
0.4	0.5	$0.4 \leqslant r_c < 0.5$	$0.4 \leqslant r_g < 0.5$	R
0.0				

Table 3.2: Decode into Original Message

Finally, we get the encrypted message "NONLINEAR" after assembling the characters.

4. Analysis

By this process, we have taken a number of characters and groups with there distinct range of probabilities which is unique to sender and receiver. Here we have used the default range of probabilities as 0.1. Hence sender and receiver can change the order of occurrence and range of probabilities for characters and groups in Table 2.1.1.1. Using single key we can only replace data bits of maximum 30 characters (as if anybody takes the key as four digit maximum number only 77 pixels is truly addressable). If anybody wants to encrypt long message using this technique, he or she can use subset of keys (using digits of key) from the original private key and different regions of image for every subset. Also if any body wants to use variable length of key it's possible, as *message_lenght ∞1 / number_of_keydigit*. Binary values generated form the textual information replaced in different nonlinear places in the image. As bits are placed any one bit in lower four bits of each R, G & B, the change of color of the modified pixels are invisible to human eye. If size of the text is less then number of pixels affected from the text will be less. At that time it will be harder to differentiate the encrypted image from the original image. If the image size is large and number of pixels is less then it will also be harder to differentiate the encrypted image from the original image [1] [2] [8].

5. Conclusion

Finally, we have used a table (Table 2.1.11) with different ranges of probabilities for distinct characters and groups which is totally unknown to the attackers (i.e. except sender and receiver). Furthermore, we have used the private key and nonlinear function technique (depending on key and bit length) to select both the pixel positions and bit position (of each R, G & B pixel) where the data will be hidden inside the image. If the key become unknown to anybody who wants to attack the information, we think, it will be quite impossible to him or her to find out the information from the watermarked image [9].

Acknowledgement:
We express our heartiest gratitude to the authority of Haldia Institute of Technology, Haldia, West Bengal, INDIA for providing resources used during the development process.

References:

[1] Vipin Tyagi and J. P. Agarwal "Digital Watermarking", in Computer Society of India on 09.26.2008.
[2] Sabyasachi Smanta, S. Kndar, Saurabh Dutta "Implementing Invisible Digital Watermarking on Image" in 'The Bulletin of Engineering and Science' ISSN: 09747176 SEPTEMBER, 2008 VOL.3| NO.2 Serial No.-06.
[3] Petitcolas, Ross J. Anderson and Markus G. Kuhn, "Information Hiding-A Survey" by Fabien A. P." in Proceedings of the IEEE, special issue on protection of multimedia content, 87(7):1062(1078, July 1999).
[4] Todd Owen and Scott Hauck "Arithmetic Compression on SPIHT Encoded Images" WEE Technical Report , Number UWEETR-2002- 0007 of 05/06/2002.
[5] Ian H. Willen, Radford m. Neal, and John G. Cleary, "Arithmetic coding for Data compression", Communications of the ACM, June 1987, Volume 30, Number 6 (Page no: 520 – 540).
[6] Paul G. Howard, Jeffrey Scott Vitter, "Practical Implementations of Arithmetic Coding" Department of Computer Science, Brown University Providence, R.I. 02912-1910.
[7] "Watermarking schemes evaluation" by Fabien A. P. Petitcolas, Microsoft Research.
[8] Paul G. Howard and Jeffrey Scott Vitter "Arithmetic Coding for Data Compression" FELLOW, IEEE.
[9] Ashok Banerjee, Ananda Mohan Ghosh, "Multimedia Technology" TMH, New Delhi.