Cryptanalysis of Digital Watermarking

Steven Aque*, Terence Brewer†, Ryan Cooper‡, Brandon Fleming§
New Mexico Institute of Mining and Technology, Socorro, New Mexico
Department of Computer Science and Engineering
*Email: steven.aque@student.nmt.edu
†Email: terence.brewer@student.nmt.edu
‡Email: ryan.cooper@student.nmt.edu
§Email: brandon.fleming@student.nmt.edu

***Abstract*—Every day, billions of pieces of digital media are created and posted online. Copyright laws exist to protect against theft, but leakers have the anonymity to continue to cause damage. How do copyright holders trace the sources of leaks? An obvious need to uniquely identify each piece of digital media without perturbing the quality arises. Watermarking fulfills this need by placing identifying information in a photo or other media that cannot be seen by the human eye. This is done by making minuscule changes in the media signal undetectable by humans and then extracting the mark from publicly-made media. A watermark may then be traced back to the person it was originally issued to, allowing the leaker to be identified and their account terminated (Traitor Tracing). To be effective, the watermark algorithms need to be robust against attacks that deter their effectiveness in identification (obfuscation) and need to be immune to watermark retrieval. The robustness of such systems will be analyzed in greater detail to see how well these watermarking methods resist obfuscation and other attacks.**

Through cryptanalysis of digital watermarking, we show that watermark algorithms are typically immune to watermark retrieval, but not immune to obfuscation and collusion attacks. However, some algorithms triumph over others in terms of robustness. Of the two algorithms investigated, Spread Spectrum was found to not be very robust against obfuscation attacks, albeit somewhat against collusion attacks. 3-level Discrete Wavelet Transform was found to be very robust against obfuscation attacks, but it is weak against ambiguity attacks, especially re-encryption. The implementation of each algorithm can be found in the appendix.

***Index Terms*—Spread Spectrum, Three Layer Discrete Wavelet Transform, Invisible Watermark, Cryptography, Obfuscation, Collusion.**

I. Introduction

In the ever-evolving online landscape, there is no greater form of expression than digital media. Be it audio, video, or images, billions of new pieces of content are being created and uploaded daily. As with any other form of property, many of these are even available for sale through various mediums. Unfortunately, digitization is not enough to escape the ever looming problem that plagues any type of product: theft. In fact, being in a digital format allows an even more significant threat to emerge, and this threat is redistribution. Even with proper copyrighting, it may be difficult to pursue legal action against content thieves. So what can one do to ease this process? What can be done to deter the content from being stolen in the first place? And perhaps most importantly, how can one determine the perpetrator of the leak and revoke their access to the content?

The answer is digital watermarking. Digital watermarking is the act of hiding a message signal within the digital media signal [1]. In the case of a digital media intellectual property holder, the message that will be hidden is some sort of identifier that the property holder can use to prove their ownership of the content. This is not the only use case for watermarks, though. Visible watermarks can be used either as a method of deterring theft or as a way of encouraging a user to purchase a premium version of the content. Invisible watermarks, on the other hand, can be used to not only prove content ownership, but even determine the source of a content leak [1].

These invisible watermarks are the focus our project, and specifically invisible watermarking on images. We research multiple algorithms that accomplish this and implement two: Spread Spectrum [2] and Three-Level Discrete Wavelet Transform [3]. We are interested in analyzing these algorithms to determine their effectiveness at embedding the watermark with minimal visual impact, as well as their robustness against various attacks. As one might expect, watermarks do not provide perfect security, and so we also implement and analyze methods of attacking the watermarks we implement.

II. Background

Digital watermarking is a well defined area of research and as such, we began this project by performing a literature review. The goal of this literature review is to understand the algorithms, applications, and attacks that are currently understood by academia and used in industry.

A. Algorithms

This section will talk about the two primary algorithms of focus in this paper, but will also mention other algorithms that were considered but not ultimately implemented

1) Spread-Spectrum Image Watermarking [2]: Spread spectrum technology has been applied in multimedia digital watermarking in the work of Cox [4]. The watermark is hidden in the frequency domain coefficients of the host, so each coefficient hides a small amount of watermark information that can not be detected at random. . [5]

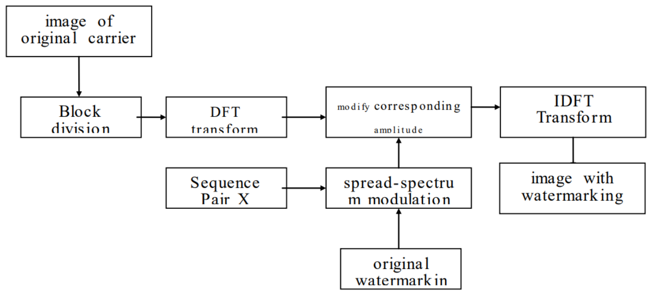


Fig. 1. Visualization of the spread spectrum process for visual embedding. Sourced from [6]

The following outlines the basic embedding and detection components of the spread spectrum watermarking algorithm on images first introduced by Cox.

Let I represent the original host image in YCrCb, and I' be the watermarked image also in YCrCb. Let M be the width of the host image and N be the height. Let k be $M * N$, that is, the count of pixels in the image. Let $W = w_1, w_2, ..., w_k$ be an k-element sequence sampled from $\mathcal{N}(0, \sigma_x/\sqrt{N})$. Let J be the watermark image to be embedded of size MxN. Let Γ be the gain defined on the watermark. Initally, let $I' = I$. Then the embedding process follows equation 1. This embedding process is also visualized at figure 1.

$$I'[:,:,0] = I[:,:,0] + \Gamma * \cdot W \cdot J \tag{1}$$

$$\mathcal{L} = \mathcal{J}_\nu(nu, Z) * pi * 0.5 * \mathcal{J}_\nu(1, \frac{0.5 * \pi * \sqrt{m^2 + n^2})}{2 * \pi * \sqrt{m^2 + n^2}}) \tag{2}$$

Given I and I', the watermark J can be easily extracted through an inverse process visualized in figure 3. Prior to simple extraction, a high-pass filter must be applied. Let w be a 21-hamming window. Let m, n both be meshgrids from -10 to 10. Let $\mathcal{J}_\nu(nu, Z)$ be the Bessel function of the first kind for each element in array Z. The low-pass filter is defined as $\mathcal{L}$ in equation. The high-pass filter to be applied is computed from $\mathcal{L}$ as initially $\mathcal{H} = -\mathcal{L}$. The value $\mathcal{H}[11, 11] = 1 - \frac{\pi * 0.5^2}{(4 * \pi)}$. Finally, the high pass filter is multiplied by $w \cdot w'$. This high pass filter is convolved with I'. Next, the noise is demodulated and applied a sign function to determine the embedded watermark J.

2) 3-Level Discrete Wavelet Transform [3]: DWT splits the image frequency into two parts: high and low frequency. The low frequency part is then split into high and low frequency.

For each level of the DWT (3), the algorithm is performed vertical first, horiziontal second. The first level yields 4 subbands: LL1, LH1, HL1, and HH2. The following levels take the previous level's LL subband as input. As a result, the second level decomposes LL1 into four more subbands: LL2, LH2, HL2, and HH2. The same occurs in the third level, where LL2 is decomposed into an additional four subbands: LL3, LH3, HL3, and HH3. The result of these three levels of

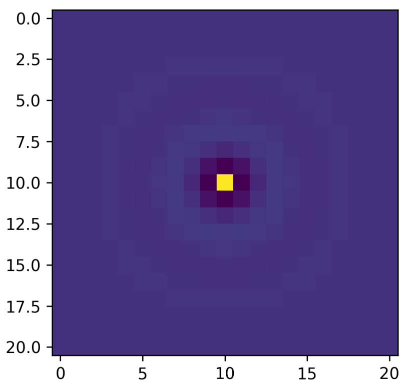


Fig. 2. Visualization of the computed high pass filter $\mathcal{H}$

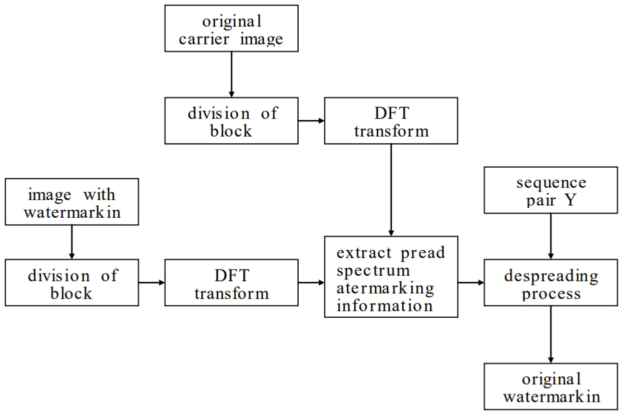


Fig. 3. Visualization of the spread spectrum process for visual extraction. Sourced from [6]

decomposition results in 10 subbands in total per component; the highest-frequency bands being LH1, HL1, and HH1 and the lowest-frequency band being LL3.

To perform the Watermark Embedding, the base image and watermark image are decomposed using 3-Level DWT into high and low frequency bands. The low frequency component of each (LL2 and WM2) are blended together with k and q scaling factors into an α Blending Embedding Technique as follows:

$$WMI = k * (LL2) + q * (WM2)$$

For Watermark Extraction, a 3-level DWT is applied to the watermarked image, decomposing it into its subbands. The watermark is then extracted using the α blending of the low-frequency approximation of the original image and watermarked image. The following formula is used to recover the watermark:

$$RW = (WMI - k * LL3)$$

This yields the low-frequency approximation of the water-

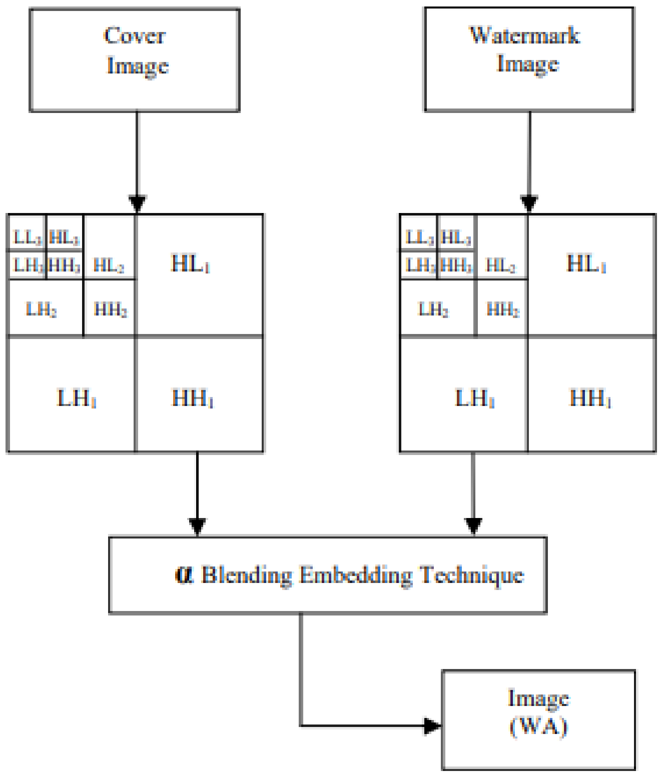


Fig. 4. Visualization of the discrete wavelet transform embedding process. Sourced from [7]

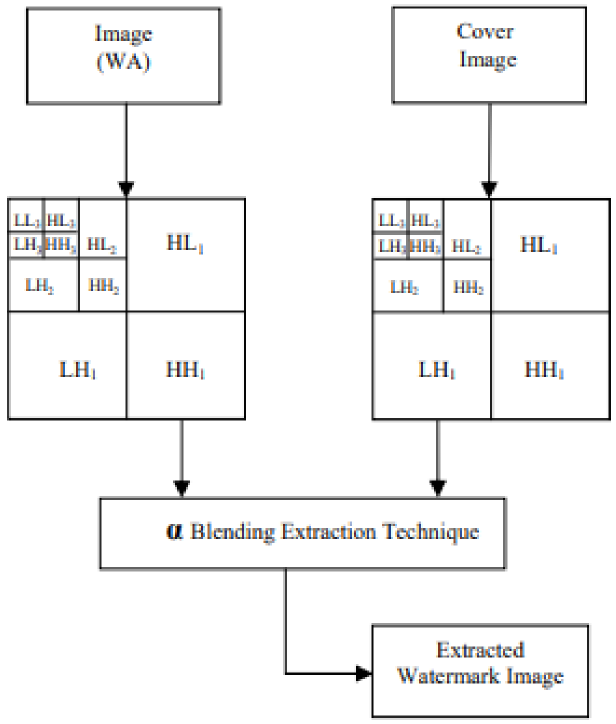


Fig. 5. Visualization of the discrete wavelet transform watermark extraction process. Sourced from [7]

mark. This equation requires knowledge of scaling factor k and the lowest-frequency portion of the original image.

3) Others: Obviously there many other algorithms to accomplish watermarking, but the aforementioned algorithms will be focused on in detail in this paper. Below are some others that were found to be used in either academia or industry, but not pursued here.

- Adaptive SVD-Based Digital Image Watermarking [8]
- Steganalysis Based on Difference Image [9]
- High Capacity Steganographic Algorithm in Color Images [10]

B. Classes of Attacks

In section 3.2 of [11], Hartung et. el define four generalized classes of attacks on digital watermarking. Class $\mathcal{A}$ attacks, or 'simple attacks' are conceptually simple attacks that "attempt to impair the embedded watermark by manipulations of the whole watermarked data", without attempting to identify or isolate the watermark. These attacks are usually quite straightforward and typically involve either linear or nonlinear filtering, addition of noise, or adding γ correction.

The second class of attacks, class $\mathcal{B}$ attacks, are known as detection-disabling attacks. These attacks attempt to break the correlation and to make the recovery of the watermark impossible or infeasible for a watermark detector, mainly through techniques like zooming, shifting, or other permutations of the image. Typically, significant modifications need to be done to the images in order for these attacks to be successful.

Class $\mathcal{C}$ attacks are known as ambiguity attacks, or attacks that attempt to confuse by producing fake original data or fake watermarked data. These ambiguity attacks attempt to mask the authentic watermark by embedding garbage data into the watermark layers of various algorithms.

Finally, class $\mathcal{D}$ attacks are known as removal attacks. These attacks attempt to analyze the watermarked data and estimate the watermark or host data in an attempt to separate the host data and the watermark, discarding the watermark. A classical example of these are collusion attacks [12].

These distinctions will be useful in quickly classifying the objective of the attacks, which will be discussed next.

C. Attacks

1) Attacks on Spread Spectrum: There are several attacks on the SS that have been named by F. Hartung et. al. [11] including addition of noise. In this method, a class $\mathcal{A}$ attack, a randomized noise matrix is generated and applied to the watermarked image. This hopes to perturb the existing watermark such that it cannot be feasibly extracted for identification, with no attempts to remove the existing watermark. Collusion attacks are a class $\mathcal{D}$ attack detailed by M. Tanha et. al. [13]. This involves the attacker obtaining a sample of the same images where averaging techniques are used to determine the values of the original image. This can then be used to remove watermarks entirely rather than obfuscating the existing watermarks. In the event the attackers are aware of the encrypting algorithm. This allows for masking (class $\mathcal{B}$)

attacks where the watermark is targeted to alter the existing watermark such that the detector is unable to identify the original. An unauthorized embedding attack can take place as well, where the attacker places a false watermark on the image. This can achieve several purposes, including the reduction of credibility from an existing work, or placing new information that confuses the detector sufficiently that the original may not be extracted.

2) Attacks on 3-Level Discrete Wavelet Transform: Several attacks exist over the DWT watermarking algorithm as are detailed by A. Samovic and J. Turan [7]. One such attack is a lossy compression attack where the image quality is lowered, which may perturb the watermark. This can be resisted by placing the watermark in the domain where quality is altered. Another method is where a random signal generation (Poisson, Gaussion, Uniform) process creates a mask the same size of the image with the maximum unnoticed strength to perturb detection of the original image. This can also happen unintentionally during Digital to Analog conversions and vice versa. Filtering attacks such as high-pass, low-pass, and Gaussian can be applied to disrupt the high frequency content found in Discrete Wavelet attacks. Collusion attacks can also be leveraged to remove the watermark from the image if enough samples are present. This evaluates the average state of pixels to remove the effect of watermarks found. This attack often requires a large sample size to be effective.

3) Generic Attacks: A common technique in any encryption-decryption architecture is the process of re-encrypting the ciphertext with meaningless data in hopes to obfuscate the original message. This technique is also applicable to digital watermarking. One such attacks is known as the N Re-Encrypts (NReE) attack. This attack involves re-encrypting the cipher image many times in hopes to obfuscate the original watermark. This technique is effective when high image gain is used when compared to the original gain used to embed the watermark originally. It is often common for an attacker to not be aware of the algorithm used in the original encryption, as such, we must also consider generalized attacks of NReE.

III. Approach

a) Survey of Existing Technology: In order to evaluate the algorithms, a sample set of images with varying watermarks will be necessary. An initial survey of existing technologies was done to determine what methods would be used for generating the sample. Very few tools were discovered, none of which suited the experiment well. A suite of watermarking tools were then created to generate sample data for attack, and to extract watermarks from the sample. Next, tools were developed to leverage some attacks against each algorithm.

b) Sample Data: For fair evaluation of performance, one image was used for sample generation. Peppers was selected for data generation (*see figure 6*) using the Bitmap format.

c) Analyzing Performance: For the purpose of performance analysis, the two primary metrics will be based on pixel difference from either the original image or watermarked image, and viewing of the generated and extracted watermark. The difference in pixel values will allow for measuring quality degradation and obfuscation of the watermark. The viewing of the watermarks will allow for evaluation of correlation between the two watermark values.

Fig. 6. Reference photo used for testing

IV. Results

A. Spread Spectrum (SS)

Obfuscation by N Re-Encrypts (NReE) will successfully obfuscate the watermark after 3 rounds, however this method affects the quality more as opposed to collusion. In *figure 8* the quality loss is displayed in conjunction to its difference from the original watermarked image. Close to 50 re-encrypts, the damage to the original is evident, with a pattern resembling the original visible in the delta of the two images. After 100 re-encrypts, severe damage has taken place to the resulting image with a likeness of the image in the delta from the original. Overall, this method of defeating the original image is fairly strong, as successful obfuscation can take place before the image is damaged. Comparing the original watermark (*figure 7*) to the extracted watermarks (*figure 8*) we can see that the watermark is successfully obfuscated after three rounds. As such running the NReE algorithm for more than three rounds is not necessary.

Another attack was the collusion attack over the SS algorithm. The results of this attack is show in figure 9. This requires multiple copies of the same image watermarked with the same noise to be successful, but can get an extremely close approximation of the original image. Using an averaging technique in the middle image, we see a very close result to the original image when iterating over a sample size of 100. The image on the right utilizes a fast Fourier transform (FFT) to approximate an original image. This method over the

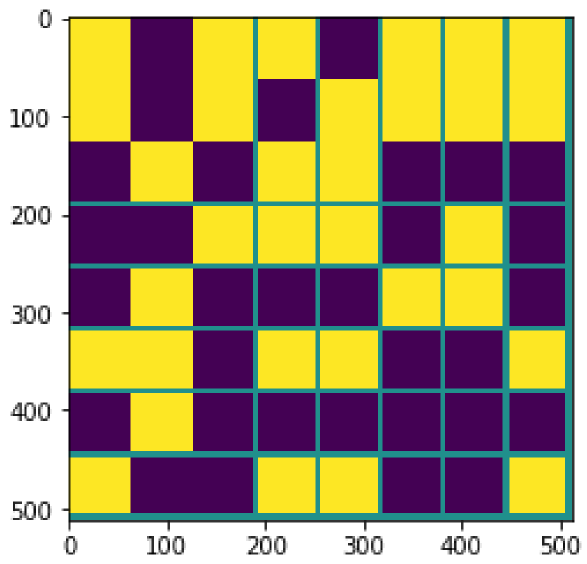


Fig. 7. SS Original watermark

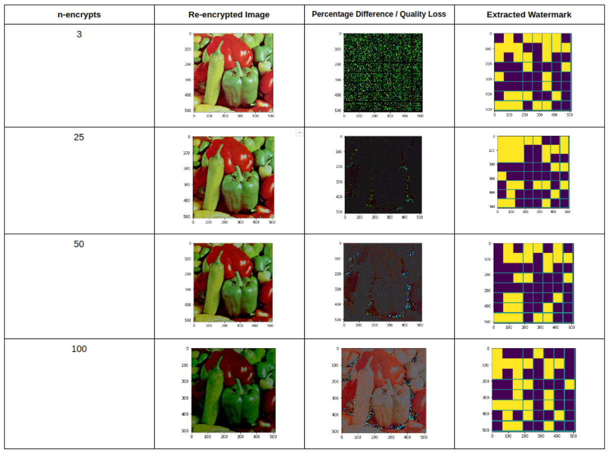

n-encrypts
Re-encrypted Image
Percentage Difference / Quality Loss
Extracted Watermark
3
25
50
100

Fig. 8. Damage to image by NReE

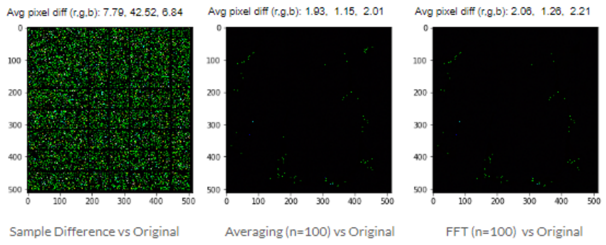


Fig. 9. Difference from original by collusion

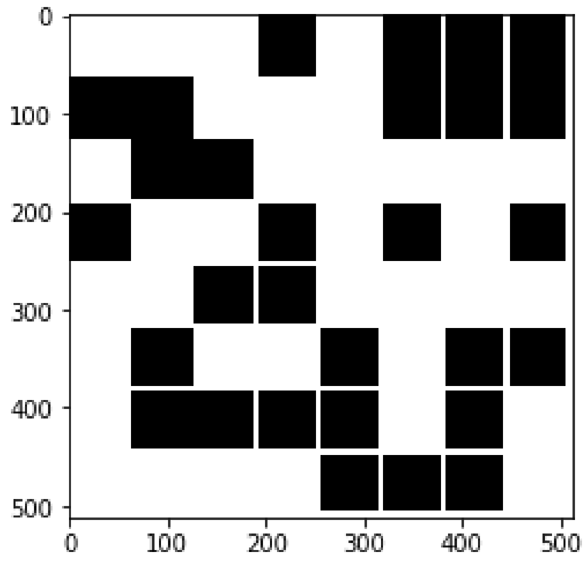


Fig. 10. Original watermark for DWT sample

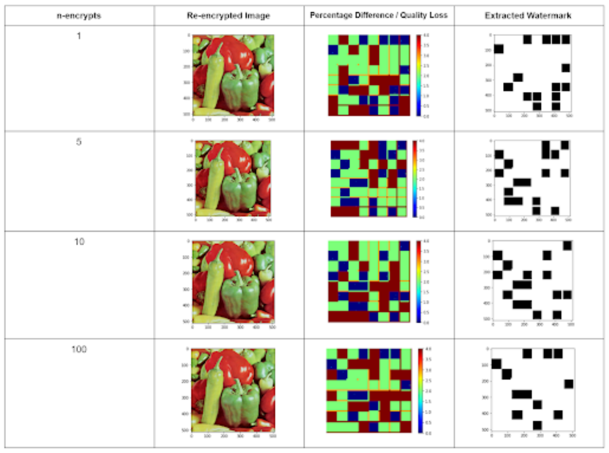

n-encrypts
Re-encrypted Image
Percentage Difference / Quality Loss
Extracted Watermark
1
5
10
100

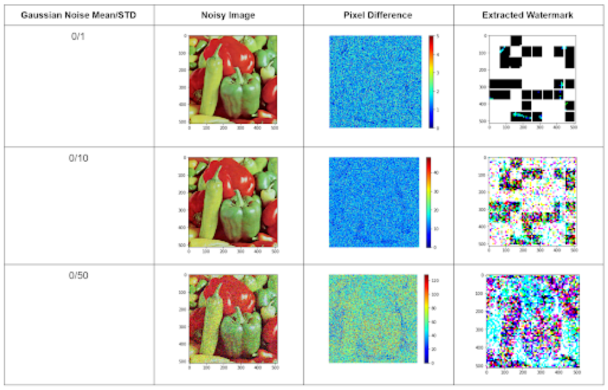
Fig. 11. Damage to image by NReE

same sample achieves a similar approximation to the averaging method.

B. Discrete Wavelet Transform (DWT)

As it performed very well over the spread spectrum algorithm, the NReE was implemented for the DWT algorithm as well. The results of NReE obfuscation is show in figure 11 This method has a few interesting differences when compared to the SS method on NReE. One difference is that even with a large value of N, no damage is perceptible to the altered image. The largest deviation from the original image is by a value of 4 in a single pixel, even over 100 rounds. This would allow for many iterations to increase destruction of the watermark. In addition, the damage to the image follows a pattern that is blocky, similar to the watermarks as opposed to the noisy pattern that resembles the attacked image. This also successfully obfuscates the original watermark after only a single round as opposed to the three rounds necessary in the SS methods.

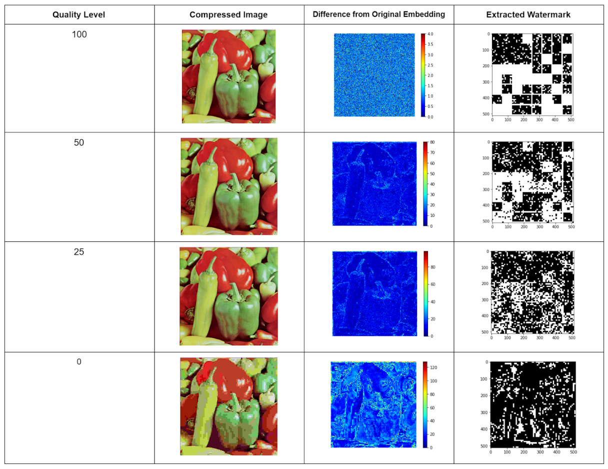
The Gaussian Noise Attack is one method that was applied to the DWT algorithm. The results of this attack are show in figure 12. In this method, a gaussian noise was generated that was then added to the watermarked image. Each noise



Gaussian Noise Mean/STD
Noisy Image
Pixel Difference
Extracted Watermark
0/1
0/10
0/50

Fig. 12. Gaussian attack on DWT

layer was tested with a mean of 0 and varying levels of standard deviation. With a standard deviation of 1 and 10, there is little damage to the image, though the watermarks are likely still usable. When a standard deviation of 50 was used, the watermark was destroyed, though the image was very obviously damaged by the operation.



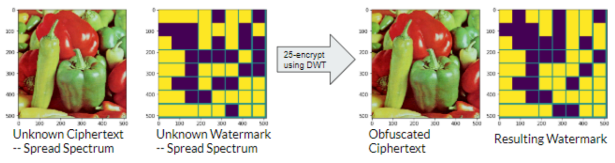
Quality Level
Compressed Image
Difference from Original Embedding
Extracted Watermark
100
50
25
0

Fig. 13. Compression attack on DWT

One method of attack on the watermarked image is to compress the image to a lower quality. The results of the attack are show in figure 13 This attack was attempted at conversions of 100% quality, 50%, 75%, 25%, and 0%. At levels 100% and 50%, the original watermark is still likely extractable and usable. At a 25% quality level, the watermark is heavily damaged but may be identified using advanced methods, though some damage to the image is obvious. When the image is compressed using a quality level of 0%, the watermark is no longer identifiable, but the image quality has suffered greatly.

C. Attack on Unknown Algorithms

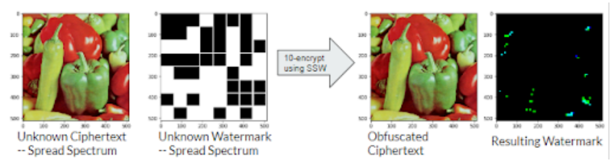
In most instances, a watermark will not be detectable by others than the original publisher. An attack will likely not allow for knowledge of the encrypting algorithm. Tests were run by attempting obfuscation by cross algorithms to determine the resistance the algorithms to obfuscation by N Re-encryptions. The first approach was to study the effects of NReE on an SS watermarked image using DWT algorithm. The results of this are shown in figure 14. This was done using 25 passes of NReE using the DWT algorithm. The resultant watermark extracted shows some change versus the original version, though several similarities are retained. There is a strong chance that the watermark could be associated with the original however.



Unknown Ciphertext -- Spread Spectrum
Unknown Watermark -- Spread Spectrum
25-encrypt using DWT
Obfuscated Ciphertext
Resulting Watermark

Fig. 14. DWT NReE attack on SS mark

Another attempt was done by applying NReE with the SS algorithm to an image watermarked by the DWT algorithm. The figure for this is listed below. The attack was run with 10 rounds using randomized watermarks and noise over a DWT marked image. By using 10 rounds, the original watermark was almost completely removed from the image. The issue remains that we do not entirely know if the watermark is destroyed, especially if the marking algorithm is unknown. As such, this attack will require a balance between alterations to the image and the quality damage that results from the SS algorithm.



Unknown Ciphertext -- Spread Spectrum
Unknown Watermark -- Spread Spectrum
10-encrypt using SSW
Obfuscated Ciphertext
Resulting Watermark

Fig. 15. SS NReE attack on DWT mark

V. Conclusions

Based on the results of the test, the algorithms are all trapdoor functions where retrieval and removal of a watermark on an image is very difficult without the key. Neither algorithm is generally immune to obfuscation or collusion. Cross encryption is not always effective, especially when using an algorithm with low impact to obfuscate one with high impact. Low impact algorithms are more prone to re-encryption from high impact algorithms however. Collusion can remove a watermark with no visible damage, however the overhead for the sample requirements will

A. Discrete Wavelet Transform

Based on the attacks perform, this algorithm is fairly robust, and circumvents the damage to the image inherent in some of the algorithms. DWT is fairly strong against most obfuscation attacks, except for the NReE attacks based on the SS or DWT algorithm, often being obfuscated after one round. Other methods show that the algorithm is rather robust, where damaging gaussian noise still fails to sufficiently obfuscate the original watermark. This was not too unexpected as images are inherently noisy and will need to be handled well. Compression was an interesting idea that was tested due to the ease use, however the DWT algorithm proved resistant to such attacks. In order to sufficiently destroy the watermark, the image was severely damaged. At levels where damage to the image was not severe, the watermark was either intact or capable of being recovered by using some advanced methods.

B. *Spread Spectrum*

As determined by the attacks performed, this algorithm is not very robust against obfuscation, though this is somewhat robust against collusion attacks. After 3 attacks, the watermark was consistently damaged enough to be destroyed and not usable. This can be done without any major damage to the image being attacked. More lightweight attacks are ineffective however as shown by the NReE by DWT attack, where 25 passes damaged a watermark, but it still may be associated with the original by advanced techniques. This is fairly robust to many attacks, requiring a balance between obfuscation and image quality.

References

[1] M. Averkiou, "Digital watermarking," 2008.

[2] A. Piper, R. Safavi-Naini, and A. Mertins, "Resolution and quality scalable spread spectrum image watermarking," 01 2005, pp. 79–90.

[3] N. Kashyap and G. R. Sinha, "Image watermarking using 3-level discrete wavelet transform (dwt)," *International Journal of Modern Education and Computer Science*, vol. 4, no. 3, p. 50–56, 2012.

[4] I. J. Cox, *Digital watermarking and steganography*. Morgan Kaufmann, 2008.

[5] T. Feng, G. Xiang, M. Lu, and L. Zhong, "A spread spectrum watermarking algorithm based on local instruction statistic," in *2013 Ninth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, Oct 2013, pp. 551–554.

[6] Z. Huang, J. Liu, Q. Lv, and H. Zhao, "Digital watermarking algorithm based on spread spectrum and fourier transform," in *2015 International Conference on Management, Education, Information and Control*. Atlantis Press, 2015/06. [Online]. Available: https://doi.org/10.2991/meici-15.2015.237

[7] A. Samovic and J. Turán, "Attacks on digital wavelet image watermarks," *Journal of Electrical Engineering*, vol. 59, pp. 131–138, 05 2008.

[8] T. T. Liu, R.Z., "Svd-based watermarking scheme for protecting rightful ownership," *IEEE Trans. on Multimedia 4(1)*, 2002.

[9] T. Zhang and X. Ping, "A new approach to reliable detection of lsb steganography in natural images," *Signal Processing*, vol. 83, pp. 2085–2093, 10 2003.

[10] S. Lyu and H. Farid, "Detecting hidden messages using higher-order statistics and support vector machines," vol. 2578, 01 2003, pp. 340–354.

[11] F. Hartung, J. Su, and B. Girod, "Spread spectrum watermarking: Malicious attacks and counterattacks," *Security and Watermarking of Multimedia Contents*, vol. 3657, 06 2000.

[12] H. Stone, "Analysis of attacks on image watermarks with randomized coefficients," NEC Research Institute, Tech. Rep., 1996.

[13] M. Tanha, D. Sajjadi, M. Abdullah, and F. Hashim, "An overview of attacks against digital watermarking and their respective countermeasures," 06 2012, pp. 265–270.

Appendix

I. Spread Spectrum Implementation

```python
from PIL import Image
import numpy as np
import cv2
import matplotlib.pyplot as plt
import os
import pickle
import math
from scipy import special, misc, ndimage, io
%matplotlib inline

def rgb2ycbcr(im):
    xform = np.array([[.299, .587, .114], [-.1687, -.3313, .5], [.5, -.4187, -.0813]])
    ycbcr = im.dot(xform.T)
    ycbcr[:,:,[1,2]] += 128
    return np.uint8(ycbcr)

def ycbcr2rgb(im):
    xform = np.array([[1, 0, 1.402], [1, -0.34414, -.71414], [1, 1.772, 0]])
    rgb = im.astype(np.float)
    rgb[:,:,[1,2]] -= 128
    rgb = rgb.dot(xform.T)
    np.putmask(rgb, rgb > 255, 255)
    np.putmask(rgb, rgb < 0, 0)
    return np.uint8(rgb)

def watermark(im, Noise, K=64, gain=1):
    A = rgb2ycbcr(np.array(im))
    B = A[:,:,0]
    M,N = B.shape
    Mb = M//K
    Nb = N//K

    plusminus1 = np.sign(np.random.randn(1,Mb*Nb))
#     plt.imshow(plusminus1.reshape(8,8))
    Watermark = np.zeros(B.shape)
    for i in range(Mb):
        for j in range(Nb):
            Watermark[i*K:(i+1)*(K-1),j*K:(j+1)*(K-1)] = plusminus1[0][i*Mb+j]
#     plt.imshow(Watermark)
#     Noise = np.round(np.random.randn(B.shape[0],B.shape[1]))
    WatermarkNoise = gain * Noise * Watermark
#     plt.imshow(WatermarkNoise)

    B = B + WatermarkNoise
    C = np.zeros((M,N,3))
    C[:,:,0] = B
    C[:,:,1] = A[:,:,1]
    C[:,:,2] = A[:,:,2]
    C = ycbcr2rgb(C)
    A = ycbcr2rgb(A)

    return (Image.fromarray(C), Watermark)

def obfuscate(n, im, K=64, gain=1):
    for i in range(n):
        noise = np.round(np.random.randn(im.size[0],im.size[0]))
        img, img_arr = watermark(im, noise)
        im = img

    return im

def decode(B, Noise, K=64):

    M,N = B.shape
    Mb = M//K
    Nb = N//K


    h = io.loadmat('h.mat')['h']

    Bconv = ndimage.convolve(B, h)

    Noise_Demod = Bconv * Noise
    Sign_Detection = np.zeros(B.shape)

    for i in range(Nb):
        for j in range(Mb):
            Sign_Detection[i*K:(i+1)*(K-1),j*K:(j+1)*(K-1)] = np.sign(sum(sum(Noise_Demod[i*K:(i+1)*(K-1),j*K:(j+1)*(K-1)])))

    return -1 * Sign_Detection

im = Image.open("pepper.bmp")
noise = np.round(np.random.randn(im.size[0],im.size[0]))
watermarked_img, watermark_arr = watermark(im, noise)
print(type(watermark_arr))
plt.imshow(watermark_arr)

# get watermarked image
plt.imshow(watermarked_img)

print(type(watermarked_img))
broken = obfuscate(10, watermarked_img)

plt.imshow(np.array(watermarked_img) - np.array(broken))

# Testing obfustication
decoded_im1 = decode(rgb2ycbcr(np.array(broken))[:,:,0], noise)
plt.imshow(decoded_im)

decoded_im2 = decode(rgb2ycbcr(np.array(watermarked_img))[:,:,0], noise)
plt.imshow(decoded_im)

np.sum(decoded_im1 == decoded_im2) / (np.sum(decoded_im1 != decoded_im2) + np.sum(decoded_im1 == decoded_im2))

pickle.dump(noise, open(directory + 'noise.p', 'wb'))
```

II. 3-DWT Implementation

```
from PIL import Image
import numpy as np
import cv2
import matplotlib.pyplot as plt
import os
import pywt
import pickle
import scipy as sp
from scipy import special, misc, ndimage, io, fftpack
import math
import pylab
%matplotlib inline

ORIGIN_RATE = 1 # q
WATERMARK_RATE = 0.009 #k

def dwt2_single(img):
    coeffs_1 = pywt.dwt2(img, 'haar', mode='reflect')
    coeffs_2 = pywt.dwt2(coeffs_1[0], 'haar', mode='reflect')
    coeffs_3 = pywt.dwt2(coeffs_2[0], 'haar', mode='reflect')
    return coeffs_1, coeffs_2, coeffs_3

def dwt2(img1, img2):
    coeffs1_1, coeffs1_2, coeffs1_3 = dwt2_single(img1)
    coeffs2_1, coeffs2_2, coeffs2_3 = dwt2_single(img2)
    return coeffs1_1, coeffs1_2, coeffs1_3, coeffs2_3

def idwt2(img, coeffs1_1_h, coeffs1_2_h, coeffs1_3_h):
    cf3 = (img, coeffs1_3_h)
    img = pywt.idwt2(cf3, 'haar', mode='reflect')

    cf2 = (img, coeffs1_2_h)
    img = pywt.idwt2(cf2, 'haar', mode='reflect')

    cf1 = (img, coeffs1_1_h)
    img = pywt.idwt2(cf1, 'haar', mode='reflect')
    return img

def embed_single_channel(orig_chan, watermark_chan):
    coeffs1_1, coeffs1_2, coeffs1_3, coeffs2_3 = dwt2(orig_chan, watermark_chan)
    embed_img = cv2.add(cv2.multiply(ORIGIN_RATE, coeffs1_3[0]), cv2.multiply(WATERMARK_RATE, coeffs2_3[0]))
    embed_img = idwt2(embed_img, coeffs1_1[1], coeffs1_2[1], coeffs1_3[1])
    np.clip(embed_img, 0, 255, out=embed_img)
    embed_img = embed_img.astype('uint8')
    return embed_img

def embed_segment(watermark, orig):
    orig_size = orig.shape[:2]
    watermark = cv2.resize(watermark, (orig_size[1], orig_size[0]))
    orig_r, orig_g, orig_b = cv2.split(orig)
    watermark_r, watermark_g, watermark_b = cv2.split(watermark)

    embed_img_r = embed_single_channel(orig_r, watermark_r)
    embed_img_g = embed_single_channel(orig_g, watermark_g)
    embed_img_b = embed_single_channel(orig_b, watermark_b)

    embed_img = cv2.merge([embed_img_r, embed_img_g, embed_img_b])
    return embed_img

def get_img_seg(image, num):
    segments = []
    if num <= 1:
        segments.append(image)
        return segments
    ratio = 1.0/float(num)
    height = image.shape[0]
    width = image.shape[1]
    pHeight = int(ratio*height)
    pHeightInterval = (height-pHeight)/(num-1)
    pWidth = int(ratio*width)
    pWidthInterval = (width-pWidth)/(num-1)

    for i in range(num):
        for j in range(num):
            x = pWidthInterval * i
            y = pHeightInterval * j
```

```python
            segments.append(image[y:y+pHeight, x:x+pWidth, :])
    return segments


def merge_img_segments(segments, num, shape):
    if num <= 1:
        return segments[0]

    ratio = 1.0/float(num)
    height =shape[0]
    width = shape[1]
    channel = shape[2]
    image = np.empty([height, width, channel], dtype=int)
    pHeight = int(ratio*height)
    pHeightInterval = (height-pHeight)/(num-1)
    pWidth = int(ratio*width)
    pWidthInterval = (width-pWidth)/(num-1)
    cnt = 0

    for i in range(num):
        for j in range(num):
            x = pWidthInterval * i
            y = pHeightInterval * j
            image[y:y+pHeight, x:x+pWidth, :] = segments[cnt]
            cnt += 1
    return image


def channel_extracting(orig_chan, embed_img_chan):
    coeffs1_1, coeffs1_2, coeffs1_3, coeffs2_3 = dwt2(orig_chan, embed_img_chan)
    extracting_img = cv2.divide(cv2.subtract(coeffs2_3[0], cv2.multiply(ORIGIN_RATE, coeffs1_3[0])), WATERMARK_RATE)
    extracting_img = idwt2(extracting_img, (None, None, None), (None, None, None), (None, None, None))
    return extracting_img

def extract_orig_segments(orig, embed_img, num):
    orig_r, orig_g, orig_b = cv2.split(orig)
    embed_img_r, embed_img_g, embed_img_b = cv2.split(embed_img)
    extracted_img_r = channel_extracting(orig_r, embed_img_r)
    extracted_img_g = channel_extracting(orig_g, embed_img_g)
    extracted_img_b = channel_extracting(orig_b, embed_img_b)
    extracting_img = cv2.merge([extracted_img_r, extracted_img_g, extracted_img_b])
    return extracting_img

def generate_watermark(shape=(512,512), K=64):
    M,N = shape
    Mb = M//K
    Nb = N//K

    plusminus1 = np.sign(np.random.randn(1,Mb*Nb))
    Watermark = np.zeros(shape)
    for i in range(Mb):
        for j in range(Nb):
            Watermark[i*K:(i+1)*(K-1),j*K:(j+1)*(K-1)] = plusminus1[0][i*Mb+j]
    Watermark_rgb = np.zeros((shape[0], shape[1], 3), dtype=np.uint8)
    for i in range(M):
        for j in range(N):
            if Watermark[i,j] == 1:
                Watermark_rgb[i,j] = np.array([0,0,0], dtype=np.uint8)
            else:
                Watermark_rgb[i,j] = np.array([255,255,255], dtype=np.uint8)

    return Watermark_rgb

def obfuscate(orig, image_segments_num, n):
    watermarks = []
    for i in range(n):
        # Generate (or load in) watermark, must be the same shape as the original
        watermark = generate_watermark(shape=orig.shape[:2])
        watermarks.append(watermark)
        # parameters
        image_segments_num = 1 # 1,2, or 4

        # encoding
        orig_segments = get_img_seg(orig, image_segments_num)
        embedding_img_segments = []
        for segment in orig_segments:
            embed_segment(watermark, segment)
            embedding_img_segments.append(embed_segment(watermark, segment))
        embed_img = merge_img_segments(embedding_img_segments, image_segments_num, orig.shape)
    return embed_img, watermarks

# load in original image to be watermarked
orig = np.array(Image.open("pepper.bmp"))
plt.imshow(orig)

# Generate (or load in) watermark, must be the same shape as the original
watermark = generate_watermark(shape=orig.shape[:2])
plt.imshow(watermark)

# parameters
image_segments_num = 1 # 1,2, or 4

# encoding
orig_segments = get_img_seg(orig, image_segments_num)
embedding_img_segments = []
for segment in orig_segments:
    embed_segment(watermark, segment)
    embedding_img_segments.append(embed_segment(watermark, segment))
embed_img = merge_img_segments(embedding_img_segments, image_segments_num, orig.shape)
plt.imshow(embed_img)

#obfuscate
broken, watermarks = obfuscate(embed_img, image_segments_num, 100)
plt.imshow(broken)

img1 = np.array(orig, dtype=np.int8)
img2 = np.array(broken, dtype=np.int8)
# Calculate the absolute difference on each channel separately
error_r = np.fabs(np.subtract(img2[:,:,0], img1[:,:,0]))
error_g = np.fabs(np.subtract(img2[:,:,1], img1[:,:,1]))
error_b = np.fabs(np.subtract(img2[:,:,2], img1[:,:,2]))
# Calculate the maximum error for each pixel
```

```
lum_img = np.array(np.maximum(np.maximum(error_r, error_g), error_b), dtype=np.uint8)
imgplot = plt.imshow(lum_img)
imgplot.set_cmap(’jet’)
plt.colorbar()
plt.axis(’off’)
pylab.show()

# decoding obfuscate
embed_img_resize = cv2.resize(broken, (orig.shape[:2][1], orig.shape[:2][0]))

orig_segments = get_img_seg(orig, image_segments_num)
embedding_img_segments = get_img_seg(embed_img_resize, image_segments_num)
extracted_img_segments = []

for i in range (0, image_segments_num*image_segments_num):
    extracted_img_segments.append(extract_orig_segments(orig_segments[i], embedding_img_segments[i], i))

extracted_watermark = np.array(merge_img_segments(extracted_img_segments, image_segments_num, orig.shape))

# clean up extraction
extracted_watermark[extracted_watermark >= 1] = 1
extracted_watermark[extracted_watermark <= 0] = 0
extracted_watermark = np.array(extracted_watermark, dtype=np.float)
plt.imshow(extracted_watermark)
broken_watermark = extracted_watermark

# decoding
embed_img_resize = cv2.resize(embed_img, (orig.shape[:2][1], orig.shape[:2][0]))

orig_segments = get_img_seg(orig, image_segments_num)
embedding_img_segments = get_img_seg(embed_img_resize, image_segments_num)
extracted_img_segments = []

for i in range (0, image_segments_num*image_segments_num):
    extracted_img_segments.append(extract_orig_segments(orig_segments[i], embedding_img_segments[i], i))

extracted_watermark = np.array(merge_img_segments(extracted_img_segments, image_segments_num, orig.shape))

# clean up extraction
extracted_watermark[extracted_watermark >= 1] = 1
extracted_watermark[extracted_watermark <= 0] = 0
extracted_watermark = np.array(extracted_watermark, dtype=np.float)
plt.imshow(extracted_watermark)
```