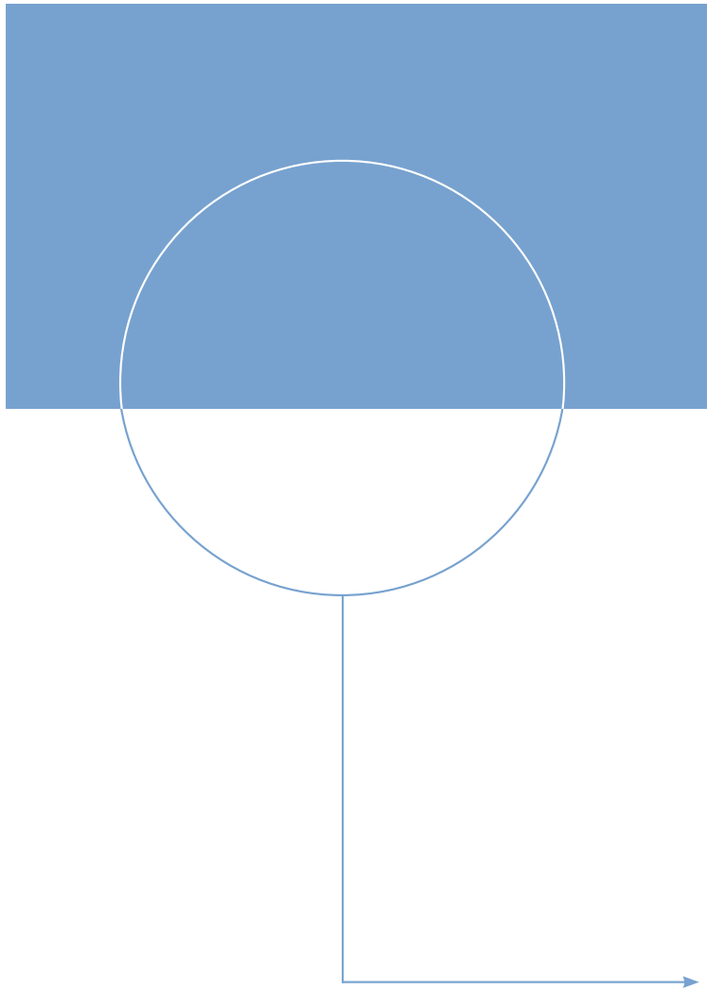


Master's thesis

2022

Kristian in't Veld

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

Master's thesis

Kristian in't Veld

Preserving Steganography Information Over Image Transformations

June 2022

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Preserving Steganography Information Over Image Transformations

Kristian in't Veld

Communication Technology and Digital Security
Submission date: June 2022
Supervisor: Colin Alexander Boyd, IIK
Co-supervisor: Bor de Kock, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Preserving Steganography Information Over Image Transformations
Student:	Kristian in't Veld

Problem description:

The field of digital image steganography has already been thoroughly studied. There exists a plethora of different methods and algorithms to use when trying to hide information in images. Most of the prior research looks at static images that are not modified at all after embedding the hidden message. Many of these steganography methods are often very susceptible to changes in the final image.

In this project, the problem of retaining information hidden in digital images after transforming the images through different common conversions is to be studied. The first goal for the project is to compare different existing steganography methods and see how well they survive different common conversions, such as resizing or file format conversion. The second goal is to look at how some of these methods can be improved to be more resilient against these types of transformations.

Date approved:	2022-01-26
Responsible professor:	Colin Alexander Boyd, IIK
Supervisor(s):	Bor de Kock, IIK

Abstract

Steganography is the art of concealing information in other mediums without making it clear that information has been hidden. In image steganography, specifically, messages can be embedded into normal looking images, but which may contain secret embedded messages that are not visible to the naked eye. Digital image steganography has been thoroughly studied for many years and there are many different methods to choose from. Normally the premise for sending these images is however that the images will not be tampered with between the sender and receiver. This might however not always be the case, if the channel for communication is limited, then one might have to use direct messaging platforms or public website to which one upload images before the recipient receives them. These third-parties might perform transformation on these images, such as down scaling, cropping, or compressing them to save disk space or for other business goals. These passive attacks will in most cases destroy any embedded steganography information that has been embedded in the images.

In this thesis we will study a selection of steganography methods, specifically, Least Significant Bit (LSB), Discrete Wavelet Transformation (DWT), Steghide, image metadata (EXIF) and file format polyglots and investigate how well they survive against such image transformations. For the methods that fail, we will also propose certain improvements that successfully are able to mitigate some of these image transformation operations.

Sammendrag

Steganografi er kunsten å skjule informasjon i andre medium. Innen bildesteganografi blir skjulte meldinger gjemt i bilder som ser normale ut for det blotte øye, men som en mottaker kan hente ut den skjulte meldingen fra. Bildesteganografi har vært studert i mange år, og det finnes mange steganografimetoder å velge mellom. Normalt er premisset en studerer innen steganografi at bilder vil bli sendt mellom to parter uten noe videre modifikasjon under overføringen, men dette er ikke nødvendigvis alltid tilfellet i dag, spesielt om muligheten for kommunikasjonskanaler mellom partene er begrenset. I slike tilfeller kan en bli tvunget til å bruke direktemeldingsplattformer eller offentlige nettsider. Disse tredjepartene kan utføre bildeopperasjoner, slik som nedskalering, beskjæring eller komprimering for å spare diskplass eller løse andre forretningsmål. Disse passive angrepene vil i de fleste tilfeller ødelegge all gjemt steganografi informasjon i bildene.

I denne oppgaven vil vi studere et utvalg steganografimetoder, spesifikt, minst signifikante bit (LSB), diskret wavelettransformasjon (DWT), Steghide, bildemetadata (EXIF) og filformatpolyglots og undersøke hvor godt de overlever mot slike bildetransformasjoner. For metodene som mislykkes, vil vi også foreslå visse forbedringer som med hell kan forbedre motstanden mot disse bildetransformasjonene.

Preface

This thesis was written during the spring of 2022 as the final chapter of my five years of studying Communications Technology and Information Security (KomTek) at the department of Information Security and Communication Technology (IIK) at the Norwegian University of Technology and Science (NTNU) in Trondheim.

The overall topic of a thesis about steganography was proposed by the institute, while I with the guidance of my supervisors formed the specific problem description and scenario to be studied during the pre-project subject of autumn 2021. A lot of effort was put into finding a problem that had not been studied too much in advance and could bring forth the shared knowledge in the field of steganography. The work on this thesis began and continued out of a period of isolation and home office caused by the Covid-19 pandemic. While this caused inconveniences and a new normal with new working patterns and solution, we at the institute of IIK were lucky enough to still be able to work from an office at campus and have some meetings before things started opening up more officially.

Even though this thesis is quite narrowed towards the field of steganography, the work of producing such a thesis and the thesis itself is meant to be a culmination of 5 years of studying to become a civil engineer in the field of information security and communication technology.

Acknowledgements

First and foremost, the work presented in this thesis would not be possible without the guidance from my supervisors, Colin Alexander Boyd and Bor de Kock, both from the Department of Information Security and Communication Technology (IIK). Thank you for your time and patience.

I would also like to direct my gratitude towards the institute staff of IIK, who, in my subjective opinion, have put more effort than any other department at NTNU into creating a good social environment for us student to study in for the past five years.

Furthermore, I would like to both thank and congratulate my fellow classmates who have joined me through the years on this journey. I appreciate the social environment we have been able to create, and thank especially those that have put the extra effort into making everyone welcome and organizing events to create the community we ended up with. I wish you all good luck with your future endeavors.

Lastly, I would like to give my thanks to all the friends and family that have supported me and made these years something special. I will never forget my time here in Trondheim.

Contents

Chapter 1
Introduction

1.1 Motivation

Steganography is the art of concealing information without making it clear that information has been hidden. One common way to achieve this is to send a non-suspicious image which contains hidden information in it. This field of research has been thoroughly studied for many decades, but usually the premise has been that the embedded image is sent as-is to the recipient without any modification during the transportation. In today's digital world, many digital platforms that offer sending or sharing of images might end up doing some sort of transformation on the image to achieve some business goal. This might be down scaling the image to save disk space, cropping the image to fit a certain dimension, like square profile pictures or converting between image formats. These transformations are in many cases destructive to any embedded steganography information that might be embedded. This project documents the current state of how well classical steganography methods are able to withstand these transformations and propose some improvements to make the methoding more robust.

1.2 Objectives

The goal of this thesis is twofold. Firstly, it aims to analyze the current performance of a selected list of steganography methods to see how well they survive common image transformations. Secondly, it aims to look at how we can improve some of these steganography methods to make them become more robust.

Figure 1.1 shows a high level overview of the imagined scenario that will be studied in this thesis. The first process shows how Alice embeds some secret message into a cover image, creating a new steganography image. Alice then sends this image to Bob, who extract this message from the image. In our scenario, however, we imagine that instead of sending the image directly to Bob, the image is instead uploaded to some website which might perform some image transformation, such

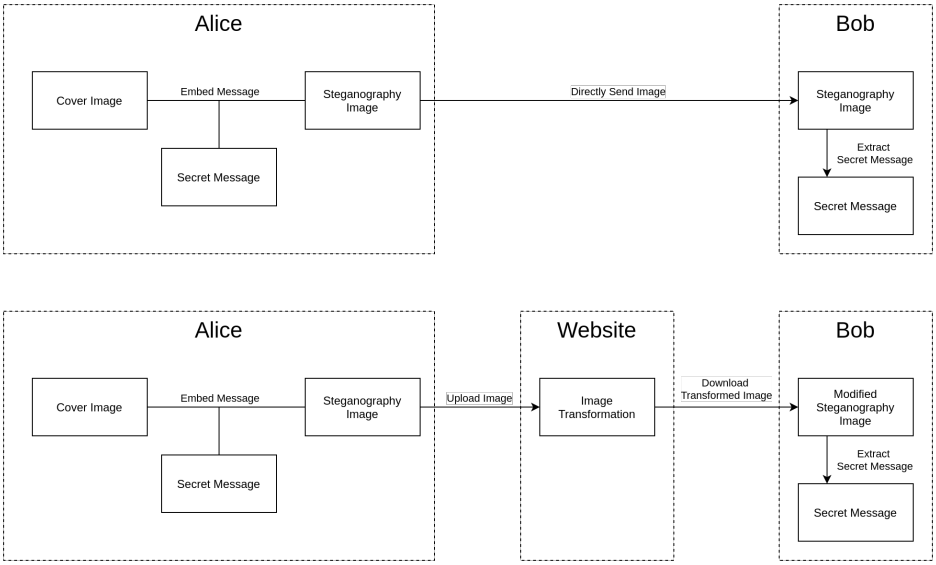


Figure 1.1: High level overview of imagined scenario to study in this thesis.

as resizing, down scaling or similar. Bob then downloads this image that has been modified by the third party website and tries to extract the embedded message. We imagine that these attacks are not active, as in, they are not aiming to destroy steganography information, but rather affects the embedded steganography as a by-product of solving some business goal by applying these transformations to the images. An imagined scenario for why one would use such services could be that only a few selected services are whitelisted in an otherwise restricted communication channel, such as communicating cross border out of an oppressive regime or similar.

To study this process, each steganography method in this thesis will be implemented first normally, then for some of the transformations we will see if we can improve them for the specific transformation in question. E.g., one method we will look into the Least Significant Bit (LSB) method, which we will describe and implementing normally, then see if we can modify this method to better survive an image transformation such as down scaling or cropping. Different strategies might have to be used for each steganography method and image transformation pair.

The image transformations in question have been selected based on them being among the most popular operations that social media websites or instant messaging platforms might apply to images uploaded by its users. Specifically, the following list will be studied:

Downscale is a common operation to reduce the resolution size of images. This is often done to reduce overall file size, or in some cases to fit a specific aspect ratio, even though in those cases the **cropping** operation might be more appropriate. An example of down scaling might be that they want to fit a certain size, so regardless of what image dimensions you upload, your image is resized to 600×600 pixels. In cases where sites only want to save size, they might just apply a non-descriminate resize where they reduce the size no matter the dimensions, or if the image is over a certain size, for instance reducing images over 1000×1000 pixels by 20%, so that a 1200×800 image becomes a 960×640 image instead.

Cropping cuts out a subsection of the original image. This is most often used when a specific aspect ratio is desired, such as when selecting a profile picture that needs to be square. For instance, if you upload an image that is 2000×2000 pixels, it might limit you to a 600×400 pixel image. Often the user is prompted to select which region is to be cut out, or it might automatically select a region, in that case, usually in the middle of the image.

Image format conversion is often used to optimize images for different target platforms (web browsers, phones, TVs, etc.). The two most common image formats on the web currently are PNG and JPEG images [Manb]. This thesis will therefor look at the conversion from PNG to JPEG and from JPEG to PNG. PNG images are lossless with a raster image format, with raw pixel data compressed using the Deflate compression algorithm [Wik22f]. JPEG on the other hand are lossy [Wik22c] and converting from PNG to JPEG often lead to minor differences. These are usually not visible, but for steganography methods that rely on pixel perfect color values, such as LSB, then this might be a problem.

To measure the performance of the different steganography methods and their improved modifications, the Peak Signal to Noise Ratio (PSNR), Mean Square Error (MSE) and Normalized Cross Correlation (NCC) will be studied. The images will be studied after they have been transformed. These methods aim to capture visual changes, while their robustness is also measured in how much of the embedded message could be extracted.

1.3 Overview and Results

This thesis will present some background material for the different steganography methods we have chosen to study. Then it will showcase how we have implemented these methods, along with presenting some improvements to mitigate certain image transformations for each steganography method. From our testing, we can see that some of our proposed improvements work against different transformations at different level of robustness and success.

Chapter 2
Background

2.1 Steganography

Traditionally, steganography has always had a strong relation to cryptography, as the two domains have much in common. In fact, one could view the art of steganography as the specific case of cryptography where the cryptography system itself is also an unknown factor instead of only having the secret key unknown, as is generally assumed for a strong cryptographic system by Kerckhoffs's principle [Wik22d]. The goal of steganography is usually to have the adversary discard the concealed message instead of putting their effort into decoding it, which might be the case for an encrypted message that obviously hides some information of importance. Looking back historically, many historic crypto systems might be considered steganography methods instead today, such as simply encoding a message in the capital letters of a cover text.

Some traditional steganography methods include invisible ink [Sel07], concatenating capital letters of a paragraph [Sel07], blinking your eyes to convey Morse code [Wik21a] or using microdots in letters [Sel07]. The latter was used extensively during WWII and the following Cold War, where a microscopic image was hidden within the dot of a punctuation. To the naked eye, the punctuation looked normal, but looking through a microscope revealed a full image, often containing a new text with the actual message [Wik21b].

We can however separate the two domains, with steganography being a way of transporting a message without having it easily detected, and cryptography makes that message hard to read. You can in fact mix them, and there is no discrete line to separate the two. For this project, we will however try to isolate steganography as only a method for embedding a secret message within a cover message. The secret message should not be trivial to spot, nor alter the cover message significantly. To achieve better security, we can always apply cryptography to the secret message before embedding it.

After we entered the digital age, steganography has seen a big transformation in appliances and usage. One is now able to hide information where it is not visible at all to the reader, such as metadata in files which is never rendered to screen. One example of this is hiding information in PNG images, where one simply appends the bytes to the end of the file. Due to how the PNG image format works, most image viewing programs will just display the original PNG image and ignore the unused appended data. Such methods are invisible to the naked eye, but are still detectable for digital systems. One could look at the file size and conclude that the file is larger than what is expected of an image of for the given dimensions, or use tools such as binwalk[1] to detect files embedded in other files in this fashion.

2.2 Steganography Methods

The following list of methods are some of the most popular steganography methods used today and also the ones that will be studied further in this project.

2.2.1 Least Significant Bit

Least Significant Bit (LSB) embeds one or more bits of information per color channel per pixel. The concept builds upon the fact that humans can usually not distinguish between minor variations of color change, so even if we have a uniform colored image where the pixels alter only by a single bit, we would usually not be able to see a difference.

One can choose an arbitrary amount of bits to use, but the more pixels one uses, the more visible the effect will be. Usually, one or two bits are chosen for images with 8-bit color depth. If one chooses to use two bits, then for an RGB image, one could embed 6 bits of data per pixel, so for a 1080×1920 image 12441600 bits or 1.4 MiB could be embedded more or less invisibly. By using existing pixels in the image, the overall size does not increase, but merely utilizes existing space in the image. The file size might increase however as in our case both JPEG and PNG utilize compression, so as a side effect, when the entropy in the image increases, it might also end up becoming harder to compress and increase the overall file size on disk for the image.

The general concept of LSB can be applied to any form of data where the least significant bits of information are not detectable by humans. Examples of this can be audio files, video files, or even the vector coordinates of scalable vector graphics.

[1] Binwalk is a tool which scans through a binary file and checks for known magic bytes [Wik22e], which indicates the start of a particular file format [ReF].

LSB is one of the most common steganography methods used by several tools such as S-Tools, Hide&Seek, Hide4GPG and Steghide [CZNY06]. The core concept of the method is simple to implement and can be adjusted easily. To avoid detection, one could choose to only use one of the bit planes, such as just the green color plane, instead of all three RGB. One could alter which pixels one chooses to use, such as following a pattern instead of just going row by row. The method is also flexible in which type of data that can be embedded, as it is just raw bytes, so any digital message can be embedded.

Some of the strength of LSB is how messages are embedded in the fine details of the image that is hard to spot, but this can also be a downside. When images are edited or transformed, these fine details are usually some of the first to be altered, such as applying color filtering, rescaling or other transformations.

2.2.2 Steghide

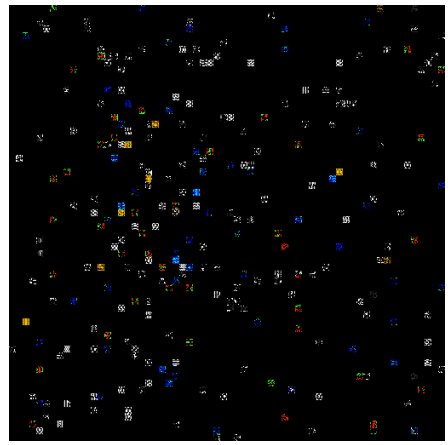
Steghide is an image steganography program written by Stefan Hetzl. It is one of the first steganography programs to make steganography easy for end users and still remains popular to this day [Het].

Steghide uses the LSB method to embed data in images, while also applying compression, encryption, check summing and methods for defending against first order statistical attacks. The program supports JPEG and BMP image formats, but also the WAV and AU audio formats. We will focus on how it works for the image formats, but the method is much the same for the audio versions.

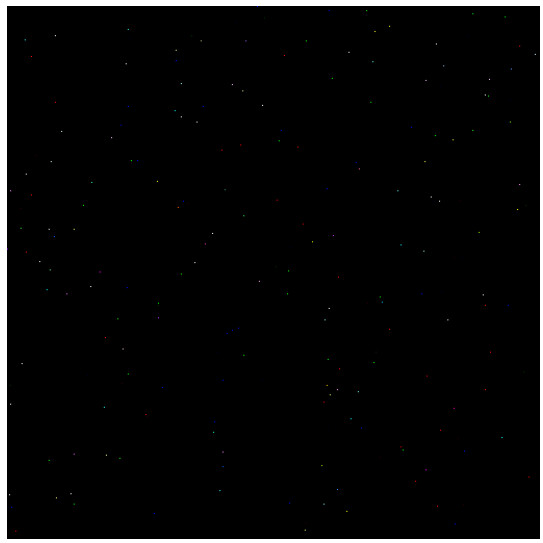
The program requires three inputs, a cover image, the secret embedded data and a password. The program starts by compressing the input data and applies by default AES-128 in CBC mode using your provided password, though other encryption methods and modes are supported. It calculates the CRC32 checksum of this encrypted data. It then embeds the encrypted data and the CRC32 checksum into the image pixels using LSB. Instead of going right to left, top to bottom, it chooses a sequence of pixels generated from a pseudo-random generator initialized with the provided password. Then, for each pixel it needs to change, it will first search the whole image for if the target pixel value already exists elsewhere. If this is the case, then it will swap the pixel values of these two pixel coordinates. If the target pixel value does not exist, then it will simply overwrite the pixel. By switching pixel values like this, the number of occurrences for each unique color value in the image remains mostly the same.

In figure 2.1 we see how Steghide embeds information into images. Lossy image formats like JPEG images can not just embed single pixels with information, so larger chunks have to be modified so to not lose details to the image compression.

(a) Lena cover image



(b) Difference between Lena.jpg and Lena-with-steghide.jpg. Data is embedded in bigger JPEG compression chunks.



(c) Difference between Lena.png and Lena-with-steghide.png. Data is embedded using LSB in individual pixel (notice small dots in the image).

Figure 2.1: Shows how Steghide hides information in the lossy JPEG and lossless PNG image formats. The difference has been amplified x50 times.

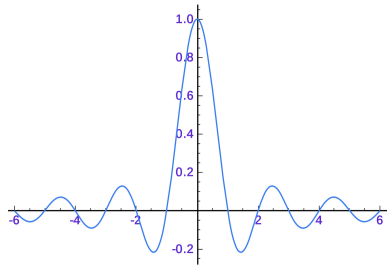


Figure 2.2: The sinc function, showing a traditional wavelet.

On lossless image formats like BMP and PNG, however, make tiny modifications to single pixels.

2.2.3 Discrete Wavelet Transformation

Wavelet transformation is a way of decomposing a signal using wavelets, much like Fourier transformation, except Fourier uses sinus and cosines waves. Discrete Wavelet Transformation (DWT), similar to Fourier transformation or Discrete Cosine Transformation (DCT) has the nice characteristics of being able to extract and emphasize the broader outlines of an image. They all work quite similarly, by trying to approximate a signal using a composition of waves or wavelets, instead of having to store the original signal itself.

Using DWT usually performs better in signals containing sharper edges due to their more spiky shape. DCT among other things used as the JPEG compression algorithm, while the new JPEG2000 has moved on to DWT [TM12].

DWT can utilize different types of wavelets, such as the traditional sinc function shown in figure 2.2. For DWT performed on images, however, the HAAR wavelet is much more common, often yielding better results in capturing the signal, as seen in figure 2.3.

Normally, DWT is applied on one dimensional signals. When doing DWT on images, one usually uses the DWT2 or 2D discrete wavelet transformation. This will analyze a 2D gray scaled image returning for matrixes, an approximation (LL), the horizontal details (LH), the vertical details (HL) and the diagonal details (HH). One can also go the other way around with the Inverse DWT (IDWT), which also has a corresponding IDWT2 function for 2D data sets. Performing DWT2 on an image and then IDWT2 will return the original image. As the DWT2 transformation returns an approximation of the image at lower scale, this is just a new image at lower resolution. One can therefor perform DWT2 on this approximation again. This

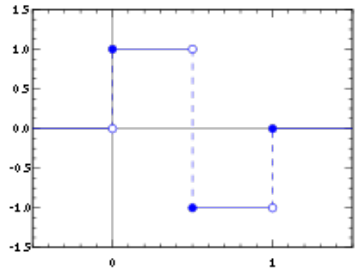


Figure 2.3: The HAAR wavelet, a popular wavelet used in DWT.

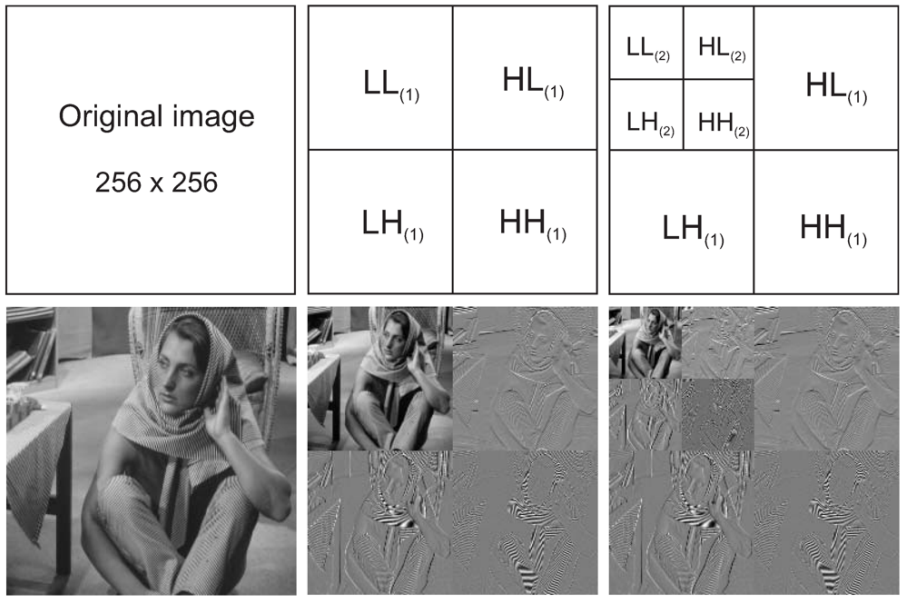


Figure 2.4: Example of 3 level DWT2 from [CG12]

can be repeated multiple times and is often referred to as N level DWT2. See figure 2.4.

The general way to embed steganography messages into images using DWT is to run the DWT2 function on both your cover image and your secret image that you are about to embed into your cover image. Then you use alpha blending between the two approximations (LL) at some level N. By using alpha blending, you select some ratio for each image, e.g., 95% cover image and 5% embedded image and add these two together. This creates a new LL approximation that you use in the IDWT2 function with the detail coefficients gained from your cover image. The outputted image should look the same as your original cover image and is your new steganography

image.

To extract the secret image, you need both the embedded image and the original cover image. Then you run DWT2 on both your original cover image and steganography image. Then you subtract the same ratio that you used on your cover image during embedding from the LL approximation of your steganography image. Multiply this by the inverse of your secret image ratio, and you should have the LL approximation of your secret image. Run the IDWT2 function with just empty LH, HL and HH details, and you should have a similar image to your original secret image, but without details.

There are two main downsides of using DWT. Firstly, both parties need the original cover image. Secondly, the extracted secret image is often not perfectly preserved, as you lose some precision in the alpha merging and by not including the original LH, HL and HH details from the secret image. Suitable secret images are therefor often simple gray scale patterns, like text, logos or QR codes.

Described above is the process for single dimension gray scale images, but it works perfectly fine for RGB images as well, just following the same process for each color plane. One does not need to embed the same image in each color plane. The images could be different, or they could embed the same image but with extra redundancy. For instance, to embed a black and white image such as an QR code inside an image one could then use the average value between red, green, and blue during reconstruction under the extraction process to decide if each pixel should be black or white, providing extra redundancy in the embedded image. Similarly, three different QR codes could be embedded for higher capacity.

2.2.4 Image Metadata

One of the easiest ways to include hidden information in images is to embed it in the image metadata. Most image formats, including PNG and JPEG, supports adding metadata, such as camera model, text description, GPS location, etc. Some of this metadata is often easily accessible and editable without extra tools, such as simple "View More" menus in image viewers or file explorers. In 2022 the .PNG Protest [FM] project started with an aim to bypass Russian censorship and spread news of the Ukraine war using images of Putin, but containing news and information in the text description of the image.

Both PNG and JPEG use the EXIF (Exchangeable image file format) which supports a key-value like structure for properties and their value. The EXIF metadata format is also widely used by other media formats, such as video and audio files. These values are often preserved by default, but external websites often scrub these to preserve the privacy of their users. For instance, sharing a photo which contains

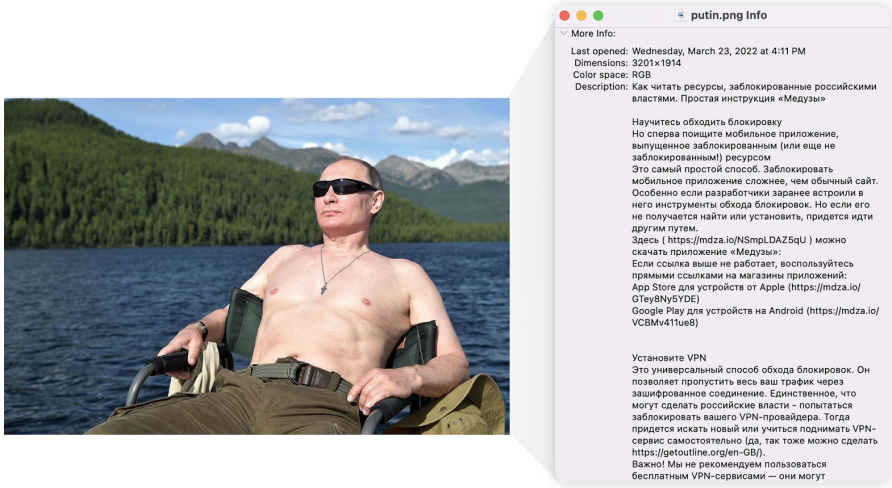


Figure 2.5: Example of image with embedded text content which is easily readable from the .PNG Protest project [FM].

geotagging with GPS coordinates could share your home or current location without your intention.

The most common option to embed data is in the *Image Description* and *User Comment* tags. These are normal plain text tags where arbitrary data can be added. There are however a plethora of other tags that could be used as well, such as *Manufacturer, Model* or *Camera Owner* to name some other text tags. One could also embed data in non-text tags more stealthily, such as in the numbers of the GPS coordinates or in time and date tags.

2.2.5 File Polyglots

A file polyglot is a file which can be interpreted in multiple ways. For instance, a PDF file which can also be opened as a ZIP file. The Mitra project [Cor] aims to document and explore how to create polyglot files between many different file formats. Another common use of this is GIF files which are also valid JavaScript files, which can be used in XSS attacks.

One common example of file polyglots is the PNG image, which is also a ZIP file. One convenient feature of the PNG image format is that one can append an arbitrary image to the end of the file, and it will just be ignored, meaning it will work perfectly fine in any image viewer or even load as an image on the web in your browser. The ZIP file however ignores any data before the magic ZIP header, which is usually at the start of the file, meaning you can prepend data to the beginning of

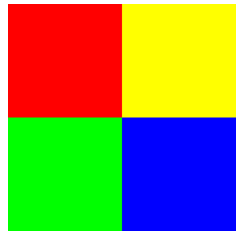


Figure 2.6: Image content of the stego.png image.

the file and the ZIP program should just ignore it. In most image or unzip programs, it should just work to open the file, though some might require renaming the file with the appropriate .zip or .png file ending.

In general, appending data to the end of files does not affect them much, and it does not have to be at the end of the file either. One could for instance slice a .wav file in half and add another file in the middle, in which case the file would sound just normal, except for in the middle where the raw data has been tampered. Extracting these files are also usually done by detecting file headers, which can be automatically done by tools such as binwalk [ReF].

```
from PIL import Image
import os

# Create small cover PNG image
img = Image.new('RGB', (2,2))
img.putpixel((0,0), (255,   0,   0))
img.putpixel((1,0), (255, 255,   0))
img.putpixel((0,1), (  0, 255,   0))
img.putpixel((1,1), (  0,   0, 255))
img.save('normal.png')

# Create a secret text file
open('secret.txt', 'w').write("Attack at Dawn")

# Create a zip file from our secret
os.system('zip secret.zip secret.txt')

# Concatinate our cover image and zip file into a new PNG file
os.system('cat normal.png secret.zip > stego.png')
```

Listing 2.1: Python script pngzip.py used to create an example 2x2 PNG image with an embedded zip file at the end.

```
$ unzip -l stego.png
Archive:  stego.png
warning [stego.png]:  71 extra bytes at beginning or within zipfile
  (attempting to process anyway)
  Length      Date    Time    Name
---------  ---------- -----   ----
       14  06-04-2022 18:37   secret.txt
---------                     -------
       14                     1 file
```

Listing 2.2: Output of "unzip -l stego.png" listing the zip file content of the PNG image.

```
$ xxd stego.png
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
00000010: 0000 0002 0000 0002 0802 0000 00fd d49a  ................
00000020: 7300 0000 0e49 4441 5478 9c63 fccf c0c0  s....IDATx.c....
00000030: 00c7 001c 0003 fe28 0384 7700 0000 0049  .......(..w....I
00000040: 454e 44ae 4260 8250 4b03 040a 0000 0000  END.B`.PK.......
00000050: 00b1 94c4 54ec a262 370e 0000 000e 0000  ....T..b7.......
00000060: 000a 001c 0073 6563 7265 742e 7478 7455  .....secret.txtU
00000070: 5409 0003 4e8a 9b62 4e8a 9b62 7578 0b00  T...N..bN..bux..
00000080: 0104 e803 0000 0464 0000 0041 7474 6163  .......d...Attac
00000090: 6b20 6174 2044 6177 6e50 4b01 021e 030a  k at DawnPK.....
000000a0: 0000 0000 00b1 94c4 54ec a262 370e 0000  ........T..b7...
000000b0: 000e 0000 000a 0018 0000 0000 0001 0000  ................
000000c0: 00a4 8100 0000 0073 6563 7265 742e 7478  .......secret.tx
000000d0: 7455 5405 0003 4e8a 9b62 7578 0b00 0104  tUT...N..bux....
000000e0: e803 0000 0464 0000 0050 4b05 0600 0000  .....d...PK.....
000000f0: 0001 0001 0050 0000 0052 0000 0000 00    .....P...R.....
```

Listing 2.3: Output of "xxd stego.png" listing the binary content of the stego.png image file.

In our script in listing 2.1 we simply create a small 2×2 PNG image with some colors which will act as our cover image. This is just a normal PNG file, and we save it to “normal.png”. We can then create a normal zip file, in this case containing just a single “secret.txt” file. If we concatenate the two files together and store it as “stego.png”, we now have a fully normal PNG image that can still act as a zip file. We can see the image in figure 2.6. Looking at the output from the “unzip -l stego.png” command in listing 2.2 we see that the unzip command detects our “secret.txt” file without any problems, though it complains about some extra bytes in the start of the file, which is the 71 bytes from our PNG image. Looking at the full binary content of the file in listing 2.3, we can see where the PNG image starts, with the PNG magic byte headers “.PNG”. We can also see where the ZIP file starts, with the magic bytes “PK”, also the file path of our “secret.txt”, along with the content of the file “Attack at Dawn”. In this case, the content of the zip file is so small that

the zip program did compress the file content. Finally, we can also see an example of the binwalk analysis of the file in listing 2.4 where binwalk first detects the magic byte headers of the PNG file, then the magic bytes of the Zlib compression data contained inside the PNG image. Then it detects the magic bytes of the zip archive file format and finally the zip file footer at the very end of the file.

```
$ binwalk stego.png
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 2 x 2, 8-bit/color RGB,
                              non-interlaced
41            0x29            Zlib compressed data, default compression
71            0x47            Zip archive data, at least v1.0 to
                              extract, compressed size: 14,
                              uncompressed size: 14, name: secret.txt
233           0xE9            End of Zip archive, footer length: 22
```

Listing 2.4: Output from the "binwalk stego.png" command.

Chapter 3
Method

3.1 Framework Implementation

To standardize testing, a python framework has been developed [iVel22]. The framework automates the process of generating testing matrices. The framework works by having a set number of input images, cover messages, implemented steganography methods and implemented image transformations. It then runs through all these and tests all combinations. This makes it easier to add new steganography methods or transformations, as the framework will handle running all the different test combinations and outputting the result.

The GitHub project contains all the 2341 generated images making up the processed results of this thesis. The project has further been developed using the Nix package manager and Pipenv for Python package locking to make the project deterministic and fully reproducible in any environment. See the full source project at [iVel22].

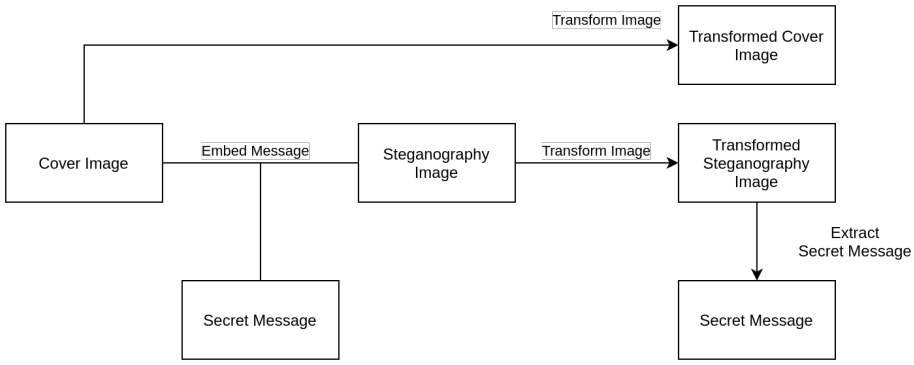


Figure 3.1: Overview of method for assessing each steganography method.

Figure 3.1 shows the overview of the method used for analyzing the different steganography methods. Firstly, the secret message will be embedded into the cover image, creating our steganography image. We will then apply the transformation to the steganography image, returning a transformed steganography image. We will then try to extract our secret message from the transformed image and compare it to the original secret message to see if we were successful or not. We will also apply our image transformation to our original cover image without the steganography method applied. We now have two images that have gone through the image transformation and these two images will then be compared using metrics such as MSE, NCC and PSNR to see how much the image was modified when we applied our steganography method to the image compared to if we had uploaded the same image without any modifications to it. We also need to compare our images after the transformation as our transformed steganography image might not be comparable to our original cover image anymore, as if the dimensions of the image has been changed (such as after down scaling or cropping) then we can not measure the difference between the images.

3.2 Choices

For many of the different steganography methods, there are many variants and choices to be made. Take for instance the LSB method. The concept itself is quite simple and basic and one have many variables to choose from, such as how many bits should be encoded, which color channels, what type of pattern should be used, etc.

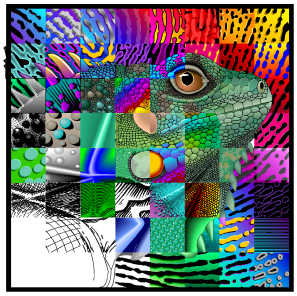
In general, the choices made when implementing these methods are usually based on a straight forward implementation or referencing prior work specifying how these options should be handled. There has been an aim to keep things as simple as possible, not adding overhead or complications that could affect the outcome of the testing and making the work easier to expand upon.

3.2.1 Images

To test the different steganography methods, test images have been provided. All images have two copies, one JPEG and one in PNG format. All the images use the RGB-color plane (no alpha plane) and with 8-bit color depth. Keeping the images standardized like this helps keeping the results comparable and simplifies the test code for the different image transformations and steganography methods.

Figure 3.2 shows the test images that have been used during testing. Some of the images, such as Clegg and Frymire are artificial images designed to be challenging to process due to high contrast and bit rate and Rainbow includes high bit rate with

(a) Clegg

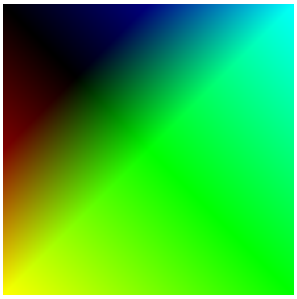


(b) Frymire

(c) Lena

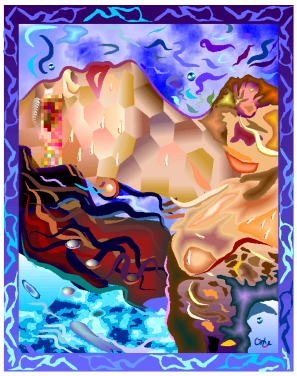
(d) Monarch

(e) Peppers



(f) Rainbow

(g) Sail



(h) Serrano

(i) Tulips

Figure 3.2: Images used during testing

difficult to compress gradients. Other images are natural photographs and should be more comparable to real world usage.

3.2.2 Secret Message & Data

The secret message embedded into the images will during the implementation be referred to as the embedded data. The aim of this project is to test the improvements on the specific steganography methods and therefor the data embedded has been viewed as pure binary data, making it more flexible and allowing any type of content to be embedded, be it text, files or other images. Since the data is only embedded as the pure binary blob that is inputted, extra steps could be taken to make the message harder to detect and extract, such as applying cryptography to the data before embedding it into the images with the steganography methods presented here.

For certain techniques, extra steps needs to be taken. The currently selected DWT based steganography method works by embedding a secret image into a cover image. There are other novel ways of embedding pure binary data using DWT, but this has not been further explored in this project. During testing, all binary data for the DWT method has therefor been encoded as QR codes. This allows arbitrary binary data to be embedded, while allowing existing libraries to apply error correction and extract the encoded message.

The EXIF metadata method is also somewhat limited in what type of data can be embedded. Per the official standard, certain EXIF tags has certain data types which it accepts, such as GPS coordinates needs to be provided as numbers, etc. We will specifically look at the EXIF tags that can embed text in them, which needs to be UTF-8 compatible, so not all arbitrary binary blobs can be directly embedded. Depending on the library used, the EXIF standard might be enforced more strictly than for other image libraries that purely ignores this part of the image. Some image libraries just ignore the whole EXIF metadata part of the image and passes it as a binary blob to the output image, while others such as the JPEG handler for the Pillow library just disregards the whole EXIF section in the output image, unless specifically enabled. For most image libraries that preserve the original EXIF metadata, however, they usually still enforce the requirements that are set for the official EXIF standard. The best we can do here is exploit the text sections, which require text to be UTF-8 encoded, meaning we do not get pure binary blobs. We can hover encode the binary data in base-85 or similar encodings, but at the cost of some encoding overhead.

3.2.3 LSB

For the LSB method, data is encoded against the two least significant bits of each color plane (red, green and blue), meaning we get 6 bits of data per pixel. We then

embed the data from right-to-left, top-to-bottom.

If one wanted a more stealthy approach, one could apply some other algorithm for which pixels to embed. An example of this is the Steghide tool, which uses a pseudo random number generator to select which pixels to embed data in. For our use case of testing the robustness during image operations, the order of pixels which are downscaled does not matter much, hence the choice to do it the trivial way.

Traditionally, LSB is applied to rasterized lossless image formats, such as LSB. This is due to the nature of LSB embedding information in the high fidelity parts of the image usually not visible to the naked eye. The JPEG compression algorithm usually smooths out these "invisible" details in the image that humans can not perceive, meaning that the LSB information is usually lost during the compression stage of the image. We also see that during our testing that LSB fails for most or all image transformations involving JPEG images.

Transformation: None

When no transformation is applied, we can just apply the normal LSB algorithm straight forward. We embed the pixels right-to-left top-to-bottom, with 2 bits per color plane leading to 6 bits of data per pixel. 2 bits gives us twice the amount of data than by using one bit while also being more or less invisible, resulting in a maximum ± 2 units of relative color value change per color plane. For the extraction we just iterate the same pixels, extracting the two last bits, concatenating them together to form the original binary blob that was encoded.

This version of the method represents the normal textbook implementation of the LSB method with no additional modifications.

Transformation: Down scaling

Our general strategy for handling down scaling is to downscale the images ourselves to the target resolution, apply the LSB method, and then upscale the modified image again to the original resolution. The goal is that when our steganography image now is transformed with down scaling, the resulting image should contain our embedded message readable as LSB data and that we should be able to use normal textbook LSB extraction to extract our message.

The pillow library supports several modes of down scaling. By default, it uses bicubic mode, which is based on cubic interpolation on all pixels that may contribute to the output value [Pil]. Another common scaling method is using the "nearest" mode, which is a simple linear scaling using only a single pixel nearest the center of the downscaled pixel. The *nearest* mode works faster than the *bicubic* mode, but

at the cost of detail, as fewer pixels are used to form the final output pixel. When upscaling using the linear method, we get a blocker, sharper image, but this has the characteristics of down scaling very closely to our target image containing our LSB embedded data. Down scaling using the default cubic interpolation leaves a blurrier image, but which at a distant look closer to our original, much like how anti-aliasing works by optimizing the viewing experience for greater distances than if you look at the image up close [Wik22a]. Similar to the Pillow image library, ImageMagick uses approximation by default when down scaling. The approximation from Pillow and ImageMagick is much harder to predict and anticipate without distorting the image too much. Another possible solution which has not been implemented, as deemed to compute heavy, would be to downscale and upscale the images repeatedly until the target modifications are present after down scaling.

Transformation: Cropping

Most image libraries handles cropping quite straight forward, simply cutting out the specified region from the original image. When cropping an image, the pixels outside the specified region gets lost. For situations where we know the region that will be cropped, we could simply apply LSB in the target area. We can however do better by applying our LSB message repeatedly over our whole image. If our message is not too long, then our message is likely to be contained once fully inside the cropped area. For longer messages, we can however split our message into short enough chunks, so that we only need to observe all our chunks at least once in the cropped area to be able to reconstruct our whole message. These chunks do not need to come in order, and if chunks near the edges of our cropped area are not complete, then as long as they are complete at least once inside the cropped area, then we are still able to extract the embedded message.

A straight forward example of this is to use the null byte ($\backslash x00$) as an escape character, then encode our chunks as *[index] [\x00] [total index] [\x00] [data...]*. We can also use double null byte to encode a null byte in our actual output. This simple scheme supports splitting a single message into 255 slices (1 indexed) with only 4 bytes of overhead per slice, and works remarkably well over our test examples.

We also need to take into account that we might start "mid-byte" when starting to scan. As a single pixel in our setup embeds 6 bits of data per pixel, we might end up reading in the middle of a byte. A byte would only start every 4 pixels if we skip horizontally. If we use the embedding technique above, we can however scan for if the image contains the expected structure from our header, if not, then append a '0' bit to the start of our extracted blob and try again. We should only have to apply this a maximum of 7 times to cycle all possible bit offsets. This strategy works no matter which configuration scheme you might choose, which affects how many bits

are encoded per bytes. In our specific example, we could apply two null bits each time as we embed an even number of bits per pixel, and should only see either 0, 2, 4 or 6 bits of from the start of a new byte from our encoded blob.

3.2.4 Steghide

Steghide is an already existing tool that has been included as a comparison for our other implemented methods. Steghide has not seen any updates since 2003, but remains to this day a robust and popular choice when novice users are looking for tools to embed stenographic secrets into images [Het].

The Steghide program supports several different media file formats, both audio and images. However, for images, it only supports BMP (**B**it**M**a**p**) and JPEG [Het]. It does not support PNG images. The BMP and PNG image formats are however very similar. Both are lossless rasterized image formats, but BMP images are usually not compressed (even though it is supported in the image format) [Wik22b]. We can also observe that when we apply Steghide with the same message and key to a BMP and JPEG image, it behaves very differently.

The reason Steghide has two different ways of embedding data in images are due to the compression that JPEG adds. We lose much more details in JPEG and need much higher redundancy when embedding our data to make sure it survives, meaning instead of changing just a couple pixels per byte, we might need to encode bigger chunks of the image to embed a single byte. The capacity in BMP images for Steghide is therefor much higher than for JPEG images.

To adapt Steghide such that we can embed information in both JPEG and PNG images, we can utilize the BMP support. As both BMP and PNG are lossless formats, we can interchange between them without losing any quality. If we want to embed information in an PNG image we can therefor just convert it to a BMP image, apply Steghide to it and then convert it back to PNG. To decode, we then just convert the PNG back to BMP and read the embedded message. Converting PNG images to BMP is usually desired, as it leaves a much higher embedding capacity than the JPEG format. This method would however also work perfectly fine with converting the PNG image to JPEG, embedding the information there and converting it back again. To combatting the PNG and JPEG conversion transformation, we can utilize this instead. When we have a PNG image that we know is going to be converted to a JPEG image, we can ourselves convert it to JPEG first, embed the secret message and convert it back to PNG before sending it off to transformation. Same if we have a JPEG image that is to be converted from JPEG to PNG, we just embed the information directly in the JPEG image as normal, but when extracting the information we convert the transformed PNG image back to JPEG to have Steghide extract the embedded message using the JPEG method instead. You can see the

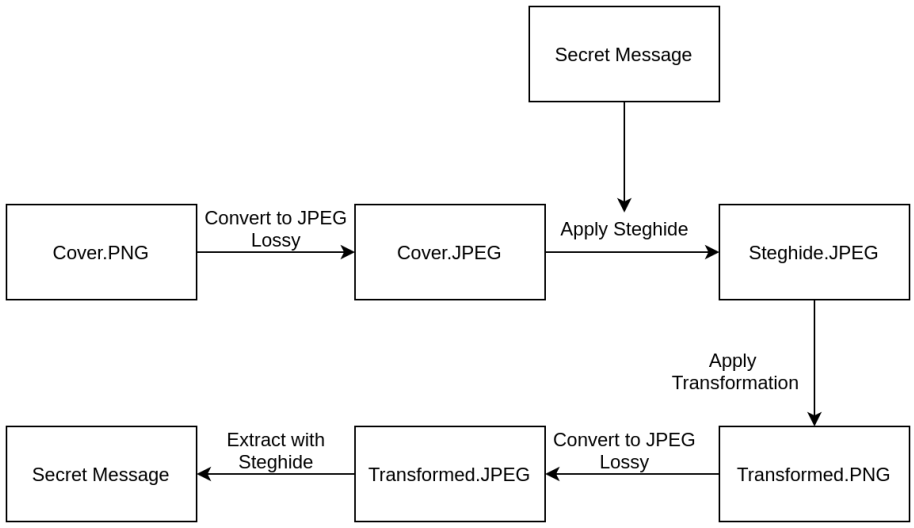


Figure 3.3: Process for mitigating the PNG to JPEG file format conversion.

difference between how Steghide embeds data in JPEG and PNG (BMP) images under the background section in figure 2.1.

Transformation: File format Conversion

For most of the methods, we do not apply any special techniques, as we want to see how well the tool handles these by itself. However, we can do a little better when it comes to file format conversion.

For conversion from PNG to JPEG we can instead of converting our PNG image to a BMP image before and embed our message, we can instead convert it to a JPEG image, loose some details, then embed our message in the JPEG image using the JPEG technique of Steghide and convert our image losslessly back to PNG. When the actual PNG to JPEG conversion happens, we then hope that the image has minimal changes to it and that we can extract the message from the final JPEG image. See figure 3.3.

For conversion from JPEG to PNG, we can do basically the same. We just embed the secret normally in our JPEG image by applying Steghide. We then send this JPEG image for transformation, which losslessly converts it to PNG. We can then again try to convert it back to JPEG and extract our secret message. See figure 3.4.

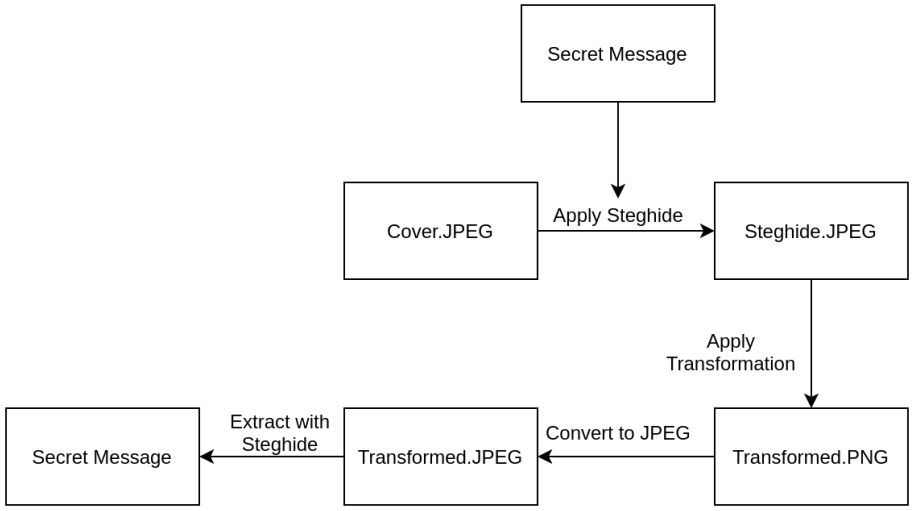


Figure 3.4: Process for mitigating the JPEG to PNG file format conversion.

3.2.5 DWT

DWT has been implemented similarly to the implementation of [KS12] and [BTA+15]. For the actual DWT implementation, the PyWavelets library has been utilized [PyW]. Following the [KS12] implementation, 3 levels of DWT2 transformation have been used. The alpha blending ratio was tweaked to 99.9% of the cover image and 0.215% of the embedded image, for maximum result with no visible effect on the final images.

Transformation: None

Using DWT, we can only embed new images. To still be able to embed binary blobs, we can just embed them as QR codes instead. QR codes offer built in error correction and redundancy. They are high contrast black and white images, making them easier to extract. For reading the QR codes, the libraries Zbar [Che], ZXing [ZXi] and DeQR [Tor] have been combined. Checking each one in order to see if any of them are able to extract the QR code message on a best effort basis. As the QR code also contains check summing, we generally receive a binary output of either being able to extract the whole message or nothing at all.

For extra redundancy, we also embed the same QR code in all red, green, and blue color planes. When decoding, we combine them into a gray scale image using the average of each of the three values. We can further convert this into a binary black and white image for easier reading for the QR code readers. QR codes contain

a mask pattern that is XOR-ed with the encoded message. The QR generator is responsible for selecting a QR mask pattern that makes the easiest to read QR code for the QR scanner, which means QR codes with high contrast and high oscillation with as few big monotone areas which could be hard to interpret for the QR scanner as to see how many pixels are contained in that area [Wik22g]. As a side effect, this also means that QR codes by design try to aim for an even distribution of black and white pixels. As we expect roughly half of our pixels to be black and half to be white, selecting the average brightness values between all the pixels in the image as the breaking point of whether a pixel should be black or white should yield a good basis for making a binary form of our image. To improve this further, we can also try a few extra values, moving the threshold up and down around the mean average value to see if any of them return a valid QR code that our scanner was able to read.

Transformation: Cropping

For the cropping transformation, we can follow a similar strategy as the one used against LSB, by repeating our QR codes across the image. This means that our QR codes become smaller and limits the amount of data that can be embedded, and we would also need a sufficiently small QR code that it fits within the cropped area. As most cropping operations often target the center area of images, repeating the QR code an odd number of times might be desirable, as then one QR code will be dead in the center. For our testing, we have chosen to repeat the QR code 3×3 times. Increasing the number of QR codes exponentially increases the space we need in the images to embed them.

3.2.6 Metadata

Embedding secret messages in the EXIF metadata has the benefit of not altering the image pixels at all. It is, however, much easier to detect and filter out. As it can become a major privacy concern to include metadata, such as geotags, many websites have started to filter out this metadata from uploaded user content. Earlier researchers have shown that even though sites such as Twitter have now stopped sharing metadata, still by using the old metadata on older images, one can still quite accurately track users [DIIP19]. Cameras and phones these days often embed more than just geotagging, they might also include tags such as phone model and name of the owner, etc. All this information can be quite sensitive and is therefor often filtered out. Seeing the dangers of including metadata in images, it is well documented that many sites have now therefor started to filter out some or all EXIF metadata embedded in images [Mana].

When implementing a scheme for embedding secret messages in the EXIF metadata, there usually are not too many variables to change, except for which EXIF tags to embed the secret message in.

To decide which tags to include, we can use the popular open source tool ExifTool [Har]. ExifTool is a common command line tool used to view and edit EXIF metadata supporting most popular media formats such as images, audio and video files. We can use this tool to set "all" tags (at least all tags the tool supports) to our text output, then open the files in Pillow or ImageMagick and check which tags have survived. We can then use any of these tags to embed our secret data.

In our implementation, we have chosen to try to set all possible tags that survived through both ImageMagick and Pillow to see which ones survive further transformations for better analysis afterwards, even though this makes more noise and increases the file size. In a more practical implementation, only one of these tags should be needed to embed the final content.

As we are working with EXIF metadata tags, we don't need to handle the different cases of transformations independently, as the EXIF metadata should survive all the transformations.

3.2.7 Polyglot

In our example, we have chosen to look at one of the oldest and simplest forms of polyglot, simply by appending our custom data to the end of the file. Similar to the EXIF metadata implementation, there is no need to handle any of the image transformations differently, as either the appended data survives the transformation or it does not.

Chapter 4
Results

The general methodology used to measure the results was to apply the normal original steganography method to an image and then apply the image transformation. We then again apply just the image transformation to the original cover image. We can then compare the two transformed images with and without steganography applied, to see how stealthy the steganography method was. We also look at how much of the secret embedded message was extracted, if any ways extracted at all. See figure 3.1.

For our implemented improved methods, we would then follow the same testing scheme, except using the modified steganography method adapted to the specific transformation applied. We would then compare this new image to the original image and look at how much of the embedded message was extracted. Finally, we can compare if the modified method did any noticeably better than the original unmodified one.

4.1 Image Formats

To test how well the different steganography methods perform, we have used JPEG and PNG images as the formats in our sample set. We have also included the image transformation of converting between these two image formats. As expected, testes related to PNG images perform better as PNG is a lossless format, and we can be more precise in handling our steganography methods. JPEG on the other hand is lossy, and we therefor do expect to see a more trouble with extracting embedded steganography messages from these images. Looking at the performance results for the different steganography methods however, it might seem that the failure rate is higher than anticipated, and for most of these cases, the reason seems to be related to the noise introduced by the JPEG format.

We could for instance expect that if we open a JPEG file and simply save it again without any modifications that the image should be saved as is. This does however not seem to be the case for Pillow and ImageMagick. The likely culprit seems to be

how both ImageMagick and Pillow handle JPEG images. When reading JPEG or any other image formats, they will convert them into a virtual pixel raster storing the full individual pixel values for the entire image, and when saving the image to the JPEG format it goes through lossy compression. This compression step very rarely outputs the same results as the original image, even if the original image was a JPEG image. In ImageMagick this is actually a documented feature as it operates as an image processor that fully decodes and re-encodes the image [Ima].

To see the effect of opening and saving JPEG images, we can use the normal 512×512 Lena image in JPEG format. We can then analyze the difference every time we read and save the image to the JPEG format using Pillow as follows:

```
from PIL import Image
import numpy as np

# Calculates the RGB value sum of the difference between two images
# It also saves an amplified version of the diff
def diff(i, img1, img2, amplify=50):
    arr1 = np.array(img1)
    arr2 = np.array(img2)

    diff = abs(arr1 - arr2)
    _sum = diff.sum()

    if amplify:
        diff = (diff.astype(np.uint64) * amplify)
        diff = diff.clip(0, 255).astype(np.uint8)

    Image.fromarray(diff).save(f'{i}-{i+1}-diff.jpg')

    return _sum

for i in range(9):
    a = Image.open(f'{i}.jpg')
    a.save(f'{i+1}.jpg')
    b = Image.open(f'{i+1}.jpg')
    print(f'Diff between image {i} and {i+1}:', diff(i, a, b))
```

Listing 4.1: Reads a JPEG image and saves it again and also outputs the difference between each iteration of saving the JPEG image to disk.

Assuming our Lena.jpg image is named “0.jpg” this script will output 8 new images and the difference between the two images. In table 4.1 we see the sum of the difference between each pixel in the images and notice that the difference starts out quite big, but that in the end, the difference seems to converge towards a fixed point. We can also visualize these changes in figure 4.1 and see that the same regions have the highest difference between each picture. Notice for instance the small white dot

Images:	Sum pixel difference:
0 and 1	89550430
1 and 2	1736428
2 and 3	285566
3 and 4	264211
4 and 5	178616
5 and 6	64011
6 and 7	18405
7 and 8	0
8 and 9	0

Table 4.1: Pixel difference for each iterative save of JPEG images using Pillow.

in image 6.jpg which is the same as the bigger red and green region in the previous images.

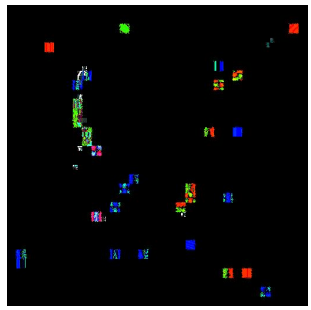
As we can see here, especially from the first comparison between 0.jpg and 1.jpg the difference can be quite big, bigger than one might expect. This is partially by design that the image should try to strip out redundant information that we humans do not notice anyway, but these big pixel differences might end up affecting our results more than initially anticipated.

4.1.1 Image Quality

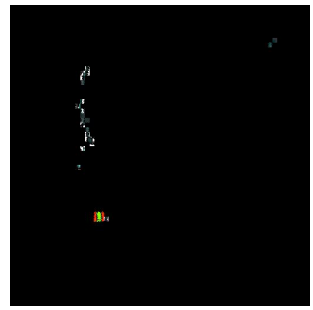
As seen above, the JPEG compression clearly affects and alters images when saved. Given the popularity of the JPEG format, these small changes are however usually acceptable, as we get to save disk space compared to storing the images in raw form. The JPEG format also allows users to specify a *quality* metric describing at what ratio keeping the image quality compared to saving disc space should be. This number usually ranges from 0 to 100 depending on the image processor and in some cases might also take code name values, such as “maximum” to set extra variables such as the chroma sub-sampling.

When compressing our PNG sample images to different JPEG qualities, we notice that the MSE seems to average between 7 and 27. Anything within this range should therefor be an acceptable error effect to an image and not be noticeable. Anything lower than 7 would mean that the operation performs better than most JPEG compressions. For PSNR, we see that the average lies between 35 and 40. Higher is better, and identical images have a score of infinite. For the NCC, the average seems to lie between 0.99224 and 0.99917. This value lies between 0 and

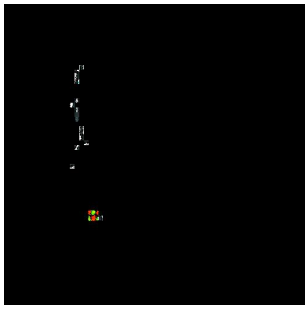
(a) Difference between 0.jpg and 1.jpg.



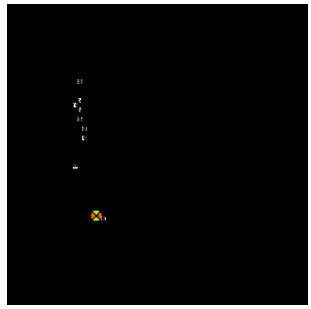
(b) Difference between 1.jpg and 2.jpg.



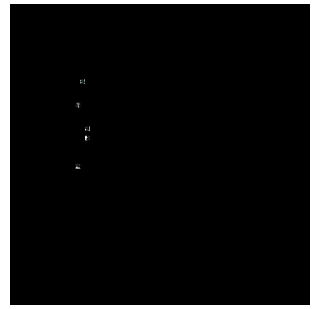
(c) Difference between 2.jpg and 3.jpg.



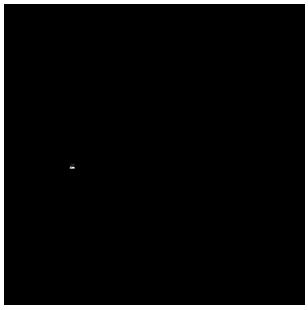
(d) Difference between 3.jpg and 4.jpg.



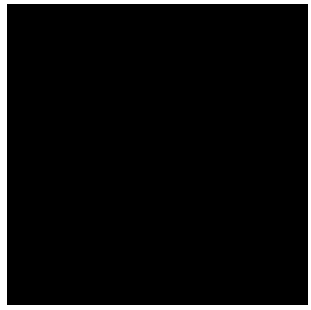
(e) Difference between 4.jpg and 5.jpg.



(f) Difference between 5.jpg and 6.jpg.



(g) Difference between 6.jpg and 7.jpg.



(h) Difference between 7.jpg and 8.jpg.

(i) Difference between 8.jpg and 9.jpg.

Figure 4.1: Visual representation of the amplified difference between iterative JPEG saves using Pillow. Color difference has been amplified x50.

1, where 1 would mean that the correlation is identical. See appendix B for more details.

As we now have some image quality metrics for just the JPEG compression algorithm, we can use these as a basis when looking further at the results of our different steganography methods. If we perform better than the JPEG compression metrics, we can quite confidently say that the methods should not alter the image too much in a noticeable way, at least not more than what users might already expect from image compression algorithms such as JPEG.

4.2 LSB

When looking at the results from the LSB method in table 4.2, we see that all tests using JPEG images fail. As mentioned in section 4.1 about image formats, the JPEG format is lossy and when we save our JPEG images after applying our LSB, not only will we lose much of the details we originally tried to embed, but we also add even more pixel errors from just not being able to save the JPEG image losslessly. We also see that transformations that involve JPEG, especially the conversion to and from JPEG, also fail, much for the same reason as mentioned above.

From 4.2 we can further see that the normal implementation for LSB works perfectly fine for all PNG images. We do, however, see that if we run our images through any of the image transformations, the message is always lost when we do not apply any improvements. Looking at our improved methods, we see that we are successfully able to embed and extract messages in images that are being cropped by both Pillow and ImageMagick with PNG images. For down scaling, our improved method is only able to handle the Fast down scaling method in the Pillow library, while the Normal down scaling of both ImageMagick and Pillow fail.

4.2.1 Transformation: Cropping

When splitting our embedded messages into smaller chunks and repeating this over our images, we see that we can easily combat cropping in lossless image formats such as PNG. This works well for both ImageMagick and Pillow as none of them are doing any more processing to the image except for straight-up cutting out the parts of the image. Our method does however introduce a slight overhead of 4 bytes per chunk and double encoding of null bytes (\x00).

4.2.2 Transformation: Down scaling

Looking at down scaling, we see full success for the Fast Pillow downs calling, but no success for the normal down scaling of Pillow or ImageMagick. We can even try to use our cropping method for repeating our message in chunks across the entire

Category	Success rate	Sample Size
No transformation PNG [Normal]	100%	9
Any transformation PNG [Normal]	0%	90
Cropping Pillow PNG [Improved]	100%	9
Cropping ImageMagick PNG [Improved]	100%	9
Down scaling Pillow Fast PNG [Improved]	100%	9
Down scaling Pillow Normal PNG [Improved]	0%	9
Down scaling ImageMagick PNG [Improved]	0%	9
Converting PNG to JPEG [Improved]	0%	9
Converting JPEG to PNG [Improved]	0%	9
No transformation JPEG [Normal]	0%	9
Any transformation JPEG [Improved]	0%	90

Table 4.2: Success rate of different LSB methods.

image, hoping that some parts, such as quiet sections of the image should stay more predictable, but still these areas of images are hard to predict for our up and down scaling.

4.2.3 Transformation: File format conversion

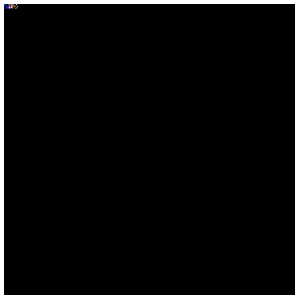
As expected, all our LSB transformations related to JPEG images fail. The JPEG format introduces slight changes, with most changes happening in the high fidelity sections such as the least significant color bits, which is where we are embedding our messages.

When converting from PNG to JPEG, we embed our message in the PNG image as normal, but this information gets destroyed by our PNG to JPEG transformation. When converting from JPEG to PNG, we first try to embed LSB messages in our JPEG image, but this step already fails, as we see from the None transformation using JPEG images.

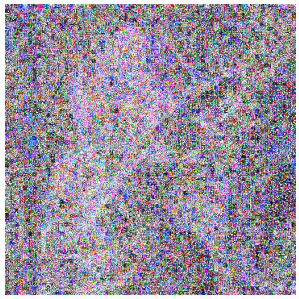
From section 4.1, we see that every time we save a JPEG image, only some regions in the image differs each time we save. This is, however, only true for the first time we save images. The reason some sections do not see any changes and that the image ends up converging to a fixed point is because some of the sections can be perfectly represented using the DCT compression. If we, however, apply LSB to these regions, then this will no longer be the case, the and these regions will be converted to their closest DCT compression representation instead.

As we see in figure 4.2 if we only embed our message a single time traditionally in

(a) Lena.png



(b) LSB with message repeated a single time in the top left corner.



(c) LSB with message repeated multiple times and split into chunks.

Figure 4.2: Illustration showing the difference between embedding our message a single time vs repeated and split into chunks. Color difference has been amplified 50 times.

the top left of our image, then only this section sees heavy re-encoding, causing big differences between our steganography image and our converted pure cover image. If we follow the same strategy as we used with cropping by repeating our message across the whole image, then we see that almost the entire image needs to be re-encoded.

4.3 Steghide

Steghide has been included as a popular tool that sees real world usage and used here as a comparison. When we look at our results, though, we see that it only survives the default usage with no transformations applied to it. We can confirm that converting the PNG to BMP images works as expected in all our tests, but the other transformations fail. Most of this is as expected, as Steghide has no built-in protection or enough redundancy to recover from operations such as cropping or destructive transformations such as resizing. One could, however, expect our mitigations against file conversions to work, but this does not seem to be the case.

For cropping and down scaling, these results are as expected. However, we had implemented mitigations against the file format conversions, but it seems like all of these have failed as well. The reason seems to be handling the JPEG format. As seen in 4.1 every time we save a JPEG image, it will become distorted to some affect.

4.3.1 Transformation: JPEG to PNG

To mitigate the JPEG to PNG transformation, we would treat the image as if it was only JPEG. We would embed the information in the image normally using Steghide

Transformation	Category	Success rate	Sample Size
No transformation JPEG	Normal	100%	9
No transformation PNG	BMP conversion	100%	9
Convert from JPEG to PNG	Improved	0%	9
Convert from PNG to JPEG	Improved	0%	9
Cropping	Normal	0%	36
Down scaling	Normal	0%	54

Table 4.3: Success rate of different Steghide methods.

when it was still the JPEG format, then the transformation would convert it into a PNG losslessly. When we then try to extract our embedded message, we would convert it back into JPEG before using Steghide for extraction. See figure 3.4 for visualization. Since we are converting a PNG image to JPEG, we lose some quality, and it seems to be enough for Steghide to fail and be unable to extract our message.

4.3.2 Transformation: PNG to JPEG

When we tried to prepare against the PNG to JPEG transformation, we would convert our PNG image to JPEG, apply Steghide and convert it back losslessly to PNG. We only lost information when converting from PNG to JPEG, but this does not matter as we have not applied our Steghide embedded message yet. However, when we send our image of to be transformed during the PNG to JPEG conversion, this step introduces quality loss before we can then try to decode our image again, so here as well Steghide fails to be able to extract any embedded message. See figure 3.3.

4.3.3 Image Quality

Looking at the metrics from table 4.4 we see that the retained image quality is quite good for normal Steghide. The way Steghide shifts around pixels as much as possible instead of straight up replacing them seems to make a difference in retaining quality and keeping the quality above average compared to JPEG conversion as seen in section 4.3. We see that our attempt at combating the file format conversion results in lower quality, party as these images are converted between PNG and JPEG in addition to the Steghide modification. Overall, Steghide seems to be a good and stealthy steganography method on both JPEG and PNG images, as long as the images are not transformed in any way.

Category	MSE	PSNR	NCC	Sample Size
Normal Steghide PNG	0.0026969	76.7852039	0.9999996	54
Normal Steghide JPEG	0.8781142	49.9698460	0.9998816	54
Convert JPEG to PNG	8.3049062	44.1299601	0.9988117	18
Convert PNG to JPEG	15.5933568	41.4259026	0.9957849	18

Table 4.4: Average MSE, PSNR and NCC for different Steghide methods.

4.4 DWT

Overall, we see that DWT performs quite well. We see little noise and effects on the image structure while being able to extract our message from difficult to combat situations such as resizing and cropping. As seen previously with the results from LSB and similar, the JPEG compression does affect us quite a bit and can be hard to combat.

4.4.1 Transformation: None

The none operation represents the default implementation of the steganography algorithm when no transformation is applied. Just out of the box, this method does not have a 100% success rate. As our implementation uses QR codes to embed the actual message, we need a certain level of quality in our output to be able to extract our message. Our embedded images are embedded into the rough structure of our image, but differentiating our image when extracting from the original structure of the image can be a challenge for images with high contrast.

Looking at figure 4.3 we can see the process of trying to extract a QR code from the Lena image. We see that the original Lena image is still somewhat visible in the combined RGB image of the extracted QR code, but when we run a filter pushing the color values to binary black and white we get out a clean and readable QR code, which can be automatically decoded to output our original binary blob that we chose to embed. Dark cover images or cover images with high contrast are however harder to extract embedded QR codes from.

Table 4.5 shows which of our sample images succeeded in extracting our embedded message. We can notice firstly that all our PNG images succeed in extracting their message. We can also notice that the JPEG images that do fail are the ones that have high contrast with lots of patterns and dark parts or are generally harder to compress, like the Rainbow.jpeg image. It might seem that the difficult to extract images make a challenge for the DWT method, but it is still able to function on its own in the PNG images, but when we in addition add the noise created from the JPEG compression, it becomes too much to handle. Looking at an example

(a) Lena cover image.

(b) Secret embedded message to embed as a QR code.

(c) DWT extract image in the red channel.

(d) DWT extract image in the green channel.

(e) DWT extract image in the blue channel.

(f) Extracted channels combined to RGB.

(g) Extracted RGB normalized to be binary black and white.

Figure 4.3: The extraction results from embedding a QR code into an image, showing the different color channels and our best effort approximation of the original QR code.

Successful:	**Failed:**
Lena.jpg	Clegg.jpg
Monarch.jpg	Frymire.jpg
Peppers.jpg	Rainbow.jpg
Sail.jpg	Serrano.jpg
Tulips.jpg	
Clegg.png	
Frymire.png	
Lena.png	
Monarch.png	
Peppers.png	
Rainbow.png	
Sail.png	
Serrano.png	
Tulips.png	

Table 4.5: Images that we could successfully extract QR codes from with no transformation interfering.

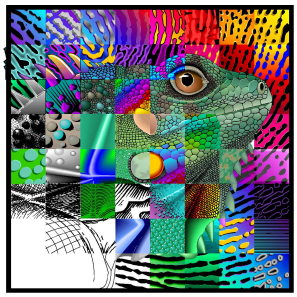
like Frymire, which is an artificial image constructed for being challenging for image processors and compression algorithms, we can see how this image goes through the DWT process in figure 4.4.

As we see in figure 4.4 the noise generated from the JPEG format is too much for us to be able to decode the extracted QR code, while the loessless PNG format does a much better job where we can see a pretty clear QR code that we are able to read and decode.

4.4.2 Transformation: Format Conversion

One of the strengths of the DWT method is that it looks at the overall structure of the image and should be fairly strong against non-visible transformations or transformations that retain a high level of structure throughout the image. DWT should therefor do quite well against file format conversions that might change the pixel values of images. It might therefor be a bit of a surprise to see how many of the tests the method fails against the file format conversion transformations. Ideally these transformations should affect the image as little as possible, but as we see in 4.6 many of these tests fails regardless.

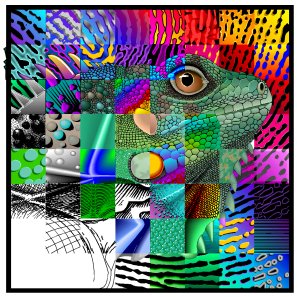
One of the downsides of using DWT is the fact that for most proposed imple-



(a) Frymire.jpg

(b) QR code extracted using DWT.

(c) Binary black and white extracted QR code.



(d) Frymire.png

(e) QR code extracted using DWT.

(f) Binary black and white extracted QR code.

Figure 4.4: Compared QR code extraction from a JPEG and PNG image.

Transformation	Category	Success rate	Sample Size
Convert JPEG to PNG	Improved	44.4%	18
Convert PNG to JPEG	Improved	11.1%	18

Table 4.6: Success rate of file format conversions with DWT.

mentations, you also need the original cover image when extracting the message. It does however seem that for this use case when looking at file format conversions, this in itself might also contribute to why these results are not performing as well as expected. When the image is converted, it has small details changed, mostly due to the JPEG compression. The original cover image however does not experience any of these changes, and using this as a detailed and absolute truth might therefor contribute to extra errors being added, compared to if the data was embedded into the DWT rough data in a stateless manner.

Transformation	Category	Success rate	Sample Size
Down scale Pillow Fast PNG	Improved	100.0%	9
Down scale Pillow Fast JPEG	Improved	0.0%	9
Down scale Pillow Normal PNG	Improved	88.9%	9
Down scale Pillow Normal JPEG	Improved	0.0%	9
Down scale ImageMagick PNG	Improved	100.0%	9
Down scale ImageMagick JPEG	Improved	0.0%	9

Table 4.7: Success rate of down scaling with DWT.

4.4.3 Transformation: Down scale

Much like the file format conversion transformation, down scaling is another transformation where we would expect DWT to perform quite well. We see that almost all of our PNG images survive and are able to extract the message from the downscale image, but again, the JPEG formats seem to be harder to decode and often failing. We do see however that are able to extract our embedded message from all library types of the down scaling, and that mostly the JPEG format is the common factor for images that fail.

Again, similar to the file format transformation, a source of error might be the difference between the cover image used for extraction and the JPEG transformations that have taken place on our steganography image.

4.4.4 Transformation: Cropping

When trying to extract messages from cropped images, we have some success. We see our method of repeating the QR code and hoping that one lands within the cropped area works well enough for us to be able to extract our embedded message, but due to the randomness of the cropping we see some randomness in which samples are able to extract a message as well.

When selecting a random area to be cropped, we might crop into an area which does not contain a complete QR code, for instance cutting a QR code in half. We can always increase the amount of QR codes we include in our image, but if the QR codes becomes too small, then their pixels blur into each other, making it harder to extract the message. The embedded capacity also decreases exponentially when we include more QR codes into our image, as the QR codes are repeated both horizontally and vertically. In addition, if the embedded message becomes too large, the QR codes will end up being highly detailed, and might not survive being embedded in the DWT frequencies.

Transformation	Category	Success rate	Sample Size
Crop Pillow PNG	Improved	11.1%	9
Crop Pillow JPEG	Improved	0.0%	9
Crop ImageMagick PNG	Improved	11.1%	9
Crop ImageMagick JPEG	Improved	0.0%	9

Table 4.8: Success rate of cropping with DWT.

Another factor to look into is the DWT2 levels. The higher level one chooses, the more robust the embedded image would end up being, but at the expense of capacity, as for every level, the resolution is halved. However, adjusting the DWT2 level for the crop transformation seems to improve our results quite a bit. As we previously were only able to extract 2 out of 36 images with the default level 3 and 3 repeated QR codes, we are able to increase this to 15 out of 21 instead by just reducing the DWT2 level from 3 to 2.

4.5 Metadata

For embedding messages in images using the EXIF metadata, the tags “Make”, “Software”, “Artist” and “Copyright” all proved effective in preserving the information for both Pillow and ImageMagick. All of these tags are also text fields where one could add what ever type of data one chooses. The method is also easily accessible by users, often not requiring any special tools to view the comments other than normal image preview applications or file browsing tools.

The selected tags were the result of checking which tags were common for the different file formats and libraries, following the example code in A. The code simply creates some test JPEG and PNG images, uses the ExifTool [Har] to set all supported EXIF tags. Then process the images with Pillow and ImageMagick and list which tags still remains on our image. From this process we get tags for each format and library and common for all 3 out of 4 of them are the “Make”, “Software”, “Artist” and “Copyright” as listed above. We do however see that Pillow drops all EXIF tags on JPEG images.

For our tests, as only the image metadata is manipulated, the images before and after are identical, so MSE and NCC are 0 with PSNR is infinite as expected.

For all the transformation using the ImageMagick library (downscale, crop, convert to JPEG and convert to PNG) the method performed perfectly. For the Pillow library, the method performed perfectly, except for when the library exported to JPEG images.

(a) Lena image. **(b)** Cropped Lena image. **(c)** Repeated pattern of QR codes to be embedded.

(d) Extracted RGB image from cropped Lena.jpg. **(e)** Normalized black and white QR code from cropped Lena.jpg.

(f) Extracted RGB image from cropped Lena.png **(g)** Normalized black and white QR code from cropped Lena.png.

Figure 4.5: The extraction process for cropped images with grid repeated patterns of QR codes used for DWT.

Transformation	Library	Success rate	Sample Size
Convert from PNG to JPEG	Pillow	0%	9
Convert from JPEG to PNG	Pillow	100%	9
All other Transformations JPEG	Pillow	0%	27
All other Transformations PNG	Pillow	100%	27
All Transformations PNG	ImageMagick	100%	27
All Transformations JPEG	ImageMagick	100%	27

Table 4.9: Success rate of EXIF metadata extraction in Pillow and ImageMagick.

Pillow operates with a model of reading files, converting these to an internal virtual raster where the pixel values are stored. Then, when an image needs to be saved those pixel values will be handed over to the file format exporter which is responsible for saving the pixel values into the correct image format, e.g., PNG or JPEG. The PNG format handler will only export certain EXIF metadata tags. It will however pass the values as is, and any non-supported EXIF tags are simply dropped. For the JPEG exporter, no EXIF metadata is saved unless this option is explicitly enabled when calling the "save" function to write the file to disk. This means that by default, every time the pillow library writes JPEG images to disk, the EXIF tags are all dropped. We can see this in the output that all the JPEG files are handled by pillow transformations (PIL in 4.9).

Another consideration for this method to keep in mind is how easy it is to detect. If one uses text input in the EXIF tags, then these are stored as plain text in both the PNG and JPEG format, so it is easy to scan for message content by just reading all the bytes of the image. It is also easy for tools to scan images for EXIF tags, as they are officially supported features of many image libraries.

This method is also highly reliant on the digital image format of the file, as it does not affect the visual aspect of the image at all, it can not be used for any sort of camera scanning or sharing. It is also not uncommon for many websites to strip all metadata tags as an extra step after doing any image transformations to protect user privacy. Any adversary that tries to look for hidden messages in images can also scan images at scale by only having to read the EXIF metadata section of the image and without having to do any sort of image pixel analysis at all.

As seen in the [FM] campaign, this method as however already been actively deployed and seen real world usage. Much of this can probably be attributed to the low entry bar of not needing specialized tools or programs to read or write the steganography messages from the images.

Transformation	Library	Success rate	Sample Size
None	None	100%	18
Any	Pillow	0%	72
Any	ImageMagick	0%	54

Table 4.10: Success rate of survived appended data on image files for Pillow and ImageMagick.

4.6 Polyglot

Same as with the EXIF metadata steganography method, the polyglot method does not affect the pixels of the image at all. Visually the images are not altered, so in use cases such as visually scanning the images to convey messages are not going to work. We can also see that both ImageMagick and Pillow fully discard the excess data at the end of the file when they re-encode the image after doing any sort of transformation on it. The only tests that succeed is the none operation, which does not alter the image at all.

This method of embedding secret content in images does however have the added benefit of being easy to implement and accessible by users, and also having virtually no restrictions to size that can be added. Most to none image viewers notice or complaint on the excess data being appended. As with the EXIF metadata method, this method can easily be scanned for, looking for redundant bytes at the end of media files.

Chapter 5
Conclusions & Further Work

In this thesis, we have looked at improvements that can be applied to different common steganography methods to make the embedded information more robust against image transformations. We have seen that in certain cases, small changes to the existing ways we embed the secret message into the images can combat different transformations that images might be put through. There does however not appear to be a golden method to apply to combat all image transformations at once, so one would be required to know which transformation one aims to protect against in advance. Further follows a summery per each steganography method.

5.1 LSB

As we saw in the results' section for the LSB method, we were especially susceptible to lossy changes and the compression of the JPEG image format. So much so that all our implementations failed, even without doing any image transformations, when we tried to save our changes to the JPEG format. The JPEG format also prevented us from successfully embedding and extracting any messages from images that were converted to or from the JPEG format. We were however successful in embedding and extracting messages without any problems for the cropping method, though at an increased noise reducing the image quality. We were also partially successful in mitigating the down scaling transformation for images downscaled using the Fast method of the Pillow library, but failed for the normal down scaling of both ImageMagick and Pillow.

Overall, the LSB method seems extra susceptible to image editing and transformations. Transformations such as tone mapping, color filters, blurring or other image effects have not been studied, but extrapolating from our testing and result we can estimate that these types of transformations will also be very destructive to the embedded LSB message. Our cropping mitigation does however seem to be one of the stronger choices among the tested improvements of all the steganography

methods, as we are able to handle most cropping dimensions and are not very reliant on getting lucky with where in the image the cropping takes place.

5.2 DWT

From the results section, we saw that DWT can survive all image transformations. It could handle both cropping, down scaling, resizing and file format conversions. It did however have a high failure rate at these and is quite image dependent. The features of the cover image are important when looking at DWT, as large regions of dark areas or having lots of contrast will interfere with the embedded cover image. We also saw especially for the cropping mitigation that our capacity is not great.

As we saw with the failure rate, though, there is room for improvements. Especially, further research into the alpha blending ratio of the cover and embedded image could be looked into further. In some images, changing the blending ratio has a larger effect than in other images, e.g., the Frymire image would often show more changes before Lena, for instance. Having a stronger embedded factor does however improve the ability to extract the image, but it also becomes easier to detect. One could for instance look into adaptive systems that choose an optimal blending factor depending on characteristics of the cover image. One could also look further into the effects of how many levels of DWT to apply, which was briefly explored in figure 4.4.

One big disadvantage of the current DWT method that has been explored in this thesis is that it requires the original cover image during extraction. It could therefor be of interest to look to alternative methods for embedding data using DWT which removes this dependency, such as the proposed method involving Arnold transformations by [SS13]. Another promising tool is the Blind Watermark python tool [Fei] which utilizes similar tactics, but with the DCT function instead of DWT. As seen by the Blind Watermark tool, it is already in a mature enough state that it can produce reliable images, surviving many active attacks on the images.

5.3 Steghide

Steghide showed great results in being able to embed messages, both with built-in encryption, check summing and easy to use interface for new users. It also showed great quality by not making too big alterations to the images. It does however not work with PNG out of the box, but our simple strategy of converting back and forth between BMP seems to work perfectly fine. The tool does however not have any built in mitigations against image tampering and failed for all the tested transformations that were applied. We did try to improve the handling of the tool to better mitigate file format conversions, but these also ended up failing, as the JPEG format introduced too much noise.

5.4 Metadata

The EXIF metadata may be the easiest and most accessible image steganography method for most users. For the ImageMagick library, it showed that it survived all image transformations for both JPEG and PNG format. For Pillow, it survived all transformations related to the PNG image format. The method does however appear to be easier to detect than most other steganography methods as well. Actors wishing to detect steganography in images could easily scan the metadata of images at scale, looking for keywords in embedded messages or suspicious encrypted data, which defeats most of the traditional intent of trying to stay stealthy with steganography. It would be more compute intensive to scan images for steganography embedded inside the actual image at scale, such as methods like LSB, Steghide or DWT, compared to looking at the different fields of the metadata.

One could however expand this method with extra steganography, such as text steganography in the embedded metadata. For instance, inside the user comment tag which we looked at in this thesis, one could utilize another level of steganography by included text embedded the with Meteor method recently proposed by [KJGR21].

It is also speculated that metadata is one of the fields that is more heavily edited and might vary from upload site and social media platform. It could also be interesting to look at how different platforms and image libraries behave, looking at which EXIF tags are usually dropped and so on. As referred to in the introduction of the metadata method, one could also embed messages in other non-text fields, such as numbers or GPS coordinates. If any such fields are still kept, while more trivial text tags are stripped out, then messages could most probably still be embedded into images and survive being uploaded to third-party websites or messaging platforms.

5.5 Polyglot

Creating file polyglots out of PNG and JPEG images is also, as shown, a simple way of embedded invisible steganography method into images. The method does however share some of the shortcomings of the metadata method. Firstly, we saw that the method failed for all transformations using the Pillow and ImageMagick libraries, as they fully re-encode the images when they save them, meaning that any data that is not used or read by the image libraries is not passed out to the transformed image. This method is also easier to scan for at scale than methods like Steghide, LSB or DWT as it is usually quite easy to check for if there is extra data appended to the end of the file, compared to having to analyze the actual pixel data to extract embedded content.

In this thesis, when the file polyglots steganography method, we have looked

at one of the simplest ways to embed data in JPEG and PNG images by simply appending byte data at the end of the file. It could, however, be interesting to look more into exploiting the image formats themselves, such as shown by in [Phi] where data was embedded into the IDAT chunk of PNG files, resulting in PNG files that could embed PHP web shells and still remain even after image resizing, as this chunk usually stays unmodified. Other fields could also be interesting to look into, such as how different image processors behave when invalid image formats are set, like editing the dimensions of PNG files to be smaller than the actual pixel image data that exists in the file.

References

[BTA+15] D. Baby, J. Thomas, *et al.*, «A novel dwt based image securing method using steganography», *Procedia Computer Science*, vol. 46, pp. 612–618, 2015, Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915001696.

[CG12] D. G. Costa and L. A. Guedes, «A discrete wavelet transform (dwt)-based energy-efficient selective retransmission mechanism for wireless image sensor networks», *Journal of Sensor and Actuator Networks*, vol. 1, no. 1, pp. 3–35, 2012. [Online]. Available: https://www.mdpi.com/2224-2708/1/1/3.

[Che] M. C. Chehab, *Zbar*. [Online]. Available: https://github.com/mchehab/zbar/tree/9f0cbc70aba627afe02cb0afdb8fbc39eb4f229d (last visited: Mar. 11, 2022).

[Cor] Corkami, *Corkami/mitra: A generator of weird files (binary polyglots, near polyglots...)* [Online]. Available: https://github.com/corkami/mitra/tree/401d133681db7f0ed9fedc10aabc5cb330131c55.

[CZNY06] M. Chen, R. Zhang, *et al.*, «Analysis of current steganography tools: Classifications & features», in *2006 International Conference on Intelligent Information Hiding and Multimedia*, IEEE, 2006, pp. 384–387.

[DIIP19] K. Drakonakis, P. Ilia, *et al.*, «Please forget where i was last summer: The privacy risks of public location (meta) data», *arXiv preprint arXiv:1901.00897*, 2019.

[Fei] G. Fei, *Python blind watermark*. [Online]. Available: https://github.com/guofei9987/blind_watermark/tree/791c91df6b3dd59e7a802eafab139505a3e682bd (last visited: May 15, 2022).

[FM] B. Farkas and T. McCauley, *PNG protests*. [Online]. Available: https://docs.google.com/document/d/1K1BKsPBmX2KvUwpYG3noGp_2NVJztZ6n9buxgF0cgd0.

[Har] P. Harvey, *ExifTool by phil harvey*. [Online]. Available: https://www.exiftool.org/.

[Het] S. Hetzl, *Steghide*. [Online]. Available: http://steghide.sourceforge.net/ (last visited: Jan. 29, 2022).

[Ima] ImageMagick, *ImageMagick Usage: JPEG Image File Format*. [Online]. Available: https://www.imagemagick.org/Usage/formats/#jpg (last visited: Feb. 5, 2022).

[iVel22] K. in't Veld, *Preserving Steganography Information Over Image Transformations: Source Code*, version v1.0.1, Jun. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6637468.

[KJGR21] G. Kaptchuk, T. M. Jois, *et al.*, «Meteor: Cryptographically secure steganography for realistic distributions», in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1529–1548.

[KS12] N. Kashyap and P. G. Sinha, «Image watermarking using 3-level discrete wavelet transform (dwt)», *International Journal of Modern Education and Computer Science*, vol. 4, Apr. 2012.

[Mana] E. M. Manifesto, *Social media sites photo metadata test results*. [Online]. Available: https://www.embeddedmetadata.org/social-media-test-results.php.

[Manb] ——, *Usage statistics of image file formats for websites*. [Online]. Available: https://w3techs.com/technologies/overview/image_format.

[Phi] Phil, *Encoding web shells in png idat chunks*. [Online]. Available: https://web.archive.org/web/20220310031346/https://www.idontplaydarts.com/2012/06/encoding-web-shells-in-png-idat-chunks/.

[Pil] Pillow, *Pillow 9.1.1 documentation: Resizing*. [Online]. Available: https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.Image.resize (last visited: Apr. 17, 2022).

[PyW] PyWavelets, *PyWavelets*. [Online]. Available: https://pywavelets.readthedocs.io/en/latest/index.html (last visited: Mar. 7, 2022).

[ReF] ReFirmLabs, *ReFirmLabs/binwalk: Firmware analysis tool*. [Online]. Available: https://github.com/ReFirmLabs/binwalk/tree/fa0c0bd59b8588814756942fe4cb5452e76c1dcd.

[Sel07] D. Sellars, «An introduction to steganography», 2007.

[SS13] P. Sehgal and V. K. Sharma, «Eliminating cover image requirement in discrete wavelet transform based digital image steganography», *International Journal of Computer Applications*, vol. 68, no. 3, 2013.

[TM12] D. Taubman and M. Marcellin, *JPEG2000 image compression fundamentals, standards and practice: image compression fundamentals, standards and practice*. Springer Science & Business Media, 2012, vol. 642. (last visited: Apr. 23, 2022).

[Tor] Torque, *Simple QR code reader*. [Online]. Available: https://github.com/torque/deqr/tree/0fbbbc565b56bb38e3901e3a5d2f76e00b8a98d7 (last visited: Mar. 11, 2022).

[Wik21a] Wikipedia contributors, *Jeremiah denton — Wikipedia, the free encyclopedia*, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Jeremiah_Denton&oldid=1054554310.

[Wik21b] ——, *Microdot — Wikipedia, the free encyclopedia*, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Microdot&oldid=1050453393.

[Wik22a] ——, *Anti-aliasing filter — Wikipedia, the free encyclopedia*, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Anti-aliasing_filter&oldid=1089603358 (last visited: Feb. 14, 2022).

[Wik22b] ——, *BMP file format — Wikipedia, the free encyclopedia*, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=BMP_file_format&oldid=1087715076 (last visited: Feb. 10, 2022).

[Wik22c] ——, *JPEG — Wikipedia, the free encyclopedia*, [Online; accessed 12-June-2022], 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=JPEG&oldid=1091175877.

[Wik22d] ——, *Kerckhoffs's principle — Wikipedia, the free encyclopedia*, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kerckhoffs%27s_principle&oldid=1087949106 (last visited: Jun. 1, 2022).

[Wik22e] ——, *List of file signatures — Wikipedia, the free encyclopedia*, [Online; accessed 12-June-2022], 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_file_signatures&oldid=1092819115.

[Wik22f] ——, *Portable network graphics — Wikipedia, the free encyclopedia*, [Online; accessed 12-June-2022], 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Portable_Network_Graphics&oldid=1090755984.

[Wik22g] ——, *QR code — Wikipedia, the free encyclopedia*, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=QR_code&oldid=1090363355 (last visited: Mar. 11, 2022).

[ZXi] ZXing, *Zxing zebra crossing barcode scannling library*. [Online]. Available: https://github.com/zxing/zxing/tree/75dbbb00dd10a5f3683553e1d56b8209b9db48e2 (last visited: Mar. 11, 2022).

Appendix

EXIF tag compatability

A short python script for checking which EXIF tags survive through being opened and saved by Pillow and ImageMagick.

```
from subprocess import check_output
from PIL import Image
import os, tempfile
import random

msg = 'Attack at dawn'
jpeg = tempfile.gettempdir() + '/' + str(random.randint(1000, 9999)) + '-exif_test.jpg'
png = tempfile.gettempdir() + '/' + str(random.randint(1000, 9999)) + '-exif_test.png'

out_jpeg_pil = jpeg[:-4] + '-pil.jpg'
out_png_pil = png[:-4] + '-pil.png'

out_jpeg_im = jpeg[:-4] + '-im.jpg'
out_png_im = png[:-4] + '-im.png'

# Create empty jpeg and png image files but with instansiated exif sections
# Here just setting the 33432=Copyright exif tag to our message
img = Image.new('RGB', (24, 24), color=(0,0,0))
exif = img.getexif()
exif.update({33432: msg})
img.save(png, exif=exif)
img.save(jpeg, exif=exif)

# Add all exif tags set to our message on the two images
check_output(['exiftool', f'-all={msg}', jpeg])
check_output(['exiftool', f'-all={msg}', png])

# Open and save the images using pillow
Image.open(jpeg).save(out_jpeg_pil)
Image.open(png).save(out_png_pil)

```

```
# Open and save the images using ImageMagick
check_output(['convert', jpeg, out_jpeg_im])
check_output(['convert', png, out_png_im])

# Check which tags survived
def filter(path):
    lines = check_output(['exiftool', path]).decode().split('\n')
    tags = [line.split(':')[0].strip() for line in lines if msg in line
    ]
    print(f'Tags for {path}: {tags}')


filter(out_jpeg_pil)
filter(out_png_pil)
filter(out_jpeg_im)
filter(out_png_im)

# Cleanup
# exiftool creates copies with the ending '_original' after editing
    files
for file in [jpeg, png, out_jpeg_pil, out_png_pil, out_jpeg_im,
    out_png_im]:
    os.remove(file)
    if os.path.isfile(file + '_original'):
        os.remove(file + '_original')
```

Listing A.1: Python script which generates some sample images with EXIF tags and processes them with Pillow and ImageMagick to see which tags survives.

Appendix B

JPEG image quality metrics

The following script was used to calculate normal MSE, PSNR and NCC metrics compared between PNG images before and after saving to the JPEG format. The script loops through all .png images in the current folder, saves the images to JPEG with different qualities and prints out some statistics.

```
import numpy as np
import math
from PIL import Image
import sys
import os

# Calculate Mean Square Error
def mse(img1, img2):
    arr1 = np.array(img1)
    arr2 = np.array(img2)
    return np.mean((arr1 - arr2)**2)

# Calculate Peak Signal to Noise Ratio
def psnr(img1, img2):
    _mse = mse(img1, img2)
    if _mse == 0:
        return float('inf')
    return 20 * math.log10(255.0 / math.sqrt(_mse))

# NCC helper function
def _normalize(arr):
    mean = np.mean(arr)
    std = np.std(arr, ddof=1)
    return (arr - mean) / std

# Calculate Normalized Cross Correlation
def ncc(img1, img2):
    arr1 = _normalize(np.array(img1))
    arr2 = _normalize(np.array(img2))
    return (1.0/(arr1.size - 1)) * np.sum(arr1 * arr2)

```

```
32
33 # Print stats of an array on a nice line with min, mean and max values
34 def stats(name, arr):
35     print(f'{name:>11}: min={np.min(arr):23.20f} mean={np.mean(arr)
       :23.20f} max={np.max(arr):23.20f}')
36
37
38 # Loop through all .png images in the current folder.
39 # Open the file and save it with JPEG quality set to 75 (default), 95,
       100 and maximum
40 # Calculate MSE, PSNR and NCC between the original PNG and JPEG image
41 # Print statistics for each metric
42 for quality in [75, 95, 100, 'maximum']:
43     MSE, PSNR, NCC, COMPRESSION = [], [], [], []
44
45     for file in os.listdir():
46         if file.endswith('.png'):
47             out = f'{file}-PIL-{quality}.jpg'
48             png = Image.open(file)
49             png.save(out, quality=quality)
50             jpg = Image.open(out)
51             MSE.append(mse(png, jpg))
52             PSNR.append(psnr(png, jpg))
53             NCC.append(ncc(png, jpg))
54             COMPRESSION.append(os.path.getsize(out)/os.path.getsize(
       file))
55
56     print('Samples:', len(MSE))
57     print(f'{quality=}')
58     stats('MSE', MSE)
59     stats('PSNR', PSNR)
60     stats('NCC', NCC)
61     stats('COMPRESSION', COMPRESSION)
62     print()
```

Listing B.1: jpeg-metrics.py

Metric:	**Min:**	**Mean:**	**Max:**
Quality:			**75**
MSE:	0.8824412027994	26.7665030346913	55.2871803104757
PSNR:	30.7045591972330	35.3791971216353	48.6739458295584
NCC:	0.9729369593636	0.9922385238503	0.9999572563902
COMPRESSION:	0.0797499530423	0.4832687166066	1.6192645623169
Quality:			**95**
MSE:	0.4099642435709	13.7919916952829	38.0083705606432
PSNR:	32.3320110930452	38.8130928731764	52.0033438097013
NCC:	0.9803209592913	0.9955972164434	0.9999797383653
COMPRESSION:	0.2268277086134	1.1097539651754	3.8115847705824
Quality:			**100**
MSE:	0.3178405761718	10.4927803360722	34.7496281922418
PSNR:	32.7213019870210	40.8877567025006	53.1087102154386
NCC:	0.9808379642100	0.9961024652709	0.9999843066327
COMPRESSION:	0.4844385445443	2.4077574082483	10.4396355353075
Quality:			**maximum**
MSE:	0.4046541849772	7.3752103014990	15.2463912776412
PSNR:	36.2991329972092	40.8270360008744	52.0599632459575
NCC:	0.9983152599554	0.9991658843794	0.9999802740820
COMPRESSION:	0.2495434843355	1.4105296332210	4.8042629352424

Table B.1: Output from jpeg-metrics.py