DYNAMIC STEGANOGRAPHY

BY SHANE DASTA

A thesis submitted to the

Graduate School—Camden

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Jean-Camille Birget

and approved by

Jean-Camille Birget

Sunil Shende

Suneeta Ramaswami

Camden, New Jersey

May 2019

ABSTRACT OF THE THESIS

Dynamic Steganography

by Shane Dasta

Thesis Director:

Jean-Camille Birget

We introduce the idea of dynamic steganography, which is essentially hiding a process within a process. We introduce components of dynamic steganography. The payload process is the process to be hidden. The cover container is the medium in which the payload process is hidden. The cover process is the process that extracts the payload process from the cover container and executes the payload process as a subprocess. We propose what an implementation could be like using image steganography and subsequently video steganography. The payload process is a set of instructions. The payload process is broken into payload components, parts of the payload process. Each payload component is hidden into a frame of the video. Upon playing the video, the payload process is extracted and executed. In the process, we go over software libraries and file types tried and used. Impediments in the implementations are encountered and proposals for future work are offered.

Dedication

I would like to thank my advisor, Dr. Jean-Camille Birget for his thoughts, criticisms, encouragements and intellectual discussions. I dedicate this thesis to my parents, Luis and Un Hui Dasta. They have supported my education for a long time with love and understanding. Lastly, to my brother and sister, Seth and Hannah Dasta. May their achievements reach beyond mine.

Table of Contents

Chapter 1

Introduction

Steganography comes from the Greek words steganós or "covered" and gráphein or "to write". Today, the term has been generalized to include any form of hidden communication. Steganography generally has two components, a *cover* and a *payload*. The cover is a distraction that hides the existence of the payload. The payload is the hidden information that is kept secret to external parties. Steganography is used to prevent detection of a secret, something that should be known only to a select number of individuals. On the other hand, cryptography is used in anticipation that presence of the secret will be detected, but its information content remains secret. The secret could be intercepted and/or modified in the absence of cryptography.

Steganography alone is only good until suspicion about the secret arises from a third party. Steganography and cryptography are complementary in improving data security. Should the steganography fail to hide the message, the cryptography will make it difficult to decipher the message. Steganography depends on avoiding suspicion; however, suspicion is difficult to measure empirically.

The field of psychology has studied suspicion, but most of the studies are not relevant to the field of steganography. Steganalysts may consider all examined material as suspicious; however, steganalysis on all examined material may be too expensive or the total amount of material may be too large to examine. For example, an intelligence agency may receive billions of images or frame data from videos shared in social media groups. Steganalysis on all the material may not be effective in detecting suspicious activity in a reasonable amount of time. We settle with the concept of *unfocused suspicion* by Schul, Burnstein, and Bardi [17], which details that the examiner, or steganalyst in our case, may receive material from multiple sources. The steganalyst

believes some or all of the material is suspicious; however, the steganalyst cannot easily determine what material contains suspicious activity. When activity in a particular material is determined suspicious, the examiner may carry out further investigation on the material yet still not know whether the material has been tampered with. We shall call this heightened awareness of suspicious activity *focused suspicion*. The concept of suspicion can be discussed further in interdisciplinary research in the future.

The art of steganography has been used long before the Information Age. For example, a Parisian slave's head was shaved and tattooed with a secret message. The secret message was hidden by having the slave's head grow out. Once the slave was delivered to the targeted recipient of the message, the slave's head would be shaved to reveal the secret. There are records of steganographic techniques dating back to the first century A.D. and to WWII, in which invisible inks made of milk, vinegar and other liquids were used by spies. During WWII, the Nazis used microdots to hide secrets in an image as small as a typed period [13, 14].

Steganography has been used in malice in the form of trojan horses, named after the Trojan War. Trojan horses are a type of rogue software, which are designed to cause harm to the target device. The trojan horse appears to have a useful purpose and acts as the *cover* while the hidden instructions in the software are used to accomplish an ulterior motive unknown to the end user of the software; the hidden instructions are known as the *payload*. An example of a trojan horse is a false anti-virus program that introduces viruses to the system. In the 1970s, Dennis Ritchie and Ken Thompson, developers of the C programming language, discovered that a trojan horse can be embedded into a compiler, which could affect routines like login for the UNIX operating system [16]. The exploitation of a login routine could be used to log the user's inputs and transmit the user's credentials to the owner of the software in order to gain access to the user's device or system.

Steganography has also been used for security and defense. Communications amongst a group of consenting parties may use steganography to exchange messages or data. These communications can be used in the military or private sector.

Image based steganography has been a popular medium to hide data. The message is the payload and the image is the cover, or distraction to transmit the payload. Methods have been contrived that make the steganography difficult to detect via human perception and computer investigation. We explore the nature of a specific set of images and the methods used to apply steganography to these images [19].

Chapter 2

Background

2.1 Image representation

Images can be represented as a bitmap (raster) or vector image. Vector images use coordinates and geometry to produce lines and curves to convey an image on the display. An image of a bowling ball could be represented by four vectors, where the bowling ball is represented by one circle and the finger holes would be represented by three ellipses, each being defined by a given radius, center point, and color. The example is oversimplified provided that the bowling ball is described with four circles, but the same image could be represented differently as a raster image.

Raster images use a grid of pixels to convey an image on a display. The pixel is the smallest unit of a raster image, representing a color, and sometimes transparency, at a particular position on the display. The bowling ball could be represented by a multitude of pixels. Pixels in one file can vary in color depth than pixels in another file. For example, the PNG file format can support a color depth of 24 bits. 8 bits are used for each component of the RGB color space: red, green and blue. While editing the binary of an image file is possible, the process becomes dependent on the file type of the image and its structure [5, 9].

Vector images provide an efficient way to represent an image via geometry and equations. For steganography, bad design is better in the sense that the raster image can contain thousands of pixels, each with 3 color components (RGB), that can carry information.

2.2 Least significant bit method

A popular method of image (and audio) based steganography is the Least significant bit (LSB) method. The LSB method embeds bits of the payload into the LSBs of the pixels of the cover image. Some variations of the method modifies the last n bits, where n is a defined number of bits. Changing the LSB of any bitstring makes the smallest possible change to the number's value [19].

2.3 Pixel indicator method

RGB image based steganography has been implemented with different methods. The pixel indicator method uses the RGB color component values to determine how many bits, if at all, are to be modified to store the hidden message. In the implementation, a color channel is selected and examined; this selected channel is hereby known as the *indicator channel*. The indicator channel is cycled among the red, green, and blue color channels. For a pixel, there is one designated indicator channel and two color channels, channel 1 and channel 2. The numbered channels are designated to be used to hide the message data if the two least significant bits (LSB) in the indicator channel are not equal to 0b00. The LSBs of channel 1 and channel 2 correspond to the two LSBs of the indicator channel as shown in Figure 2.1. For example, if the first of the two LSB bits in the indicator channel is turned on, then 2 bits of the message will be stored in channel 1. The method is good at minimizing the amount of pixels that are changed in total because some pixels will store up to a maximum of 4 bits of data from the message. Within an instance of reading from a designated indicator channel, there is a 75% chance that at least two bits from the message will be written into the qualifying channel. The method sought to improve the capacity of data that could be hidden in an image [13].

The pixel indicator technique improved capacity and hid the message in multiple channels of the RGB color space; however, it is limited to particular images. Images with pixels that have LSBs ending in 0b00 will not store the hidden message. An image that consists of only white pixels with a hex value of 0x000000 will not store any data

Indicator	Channel 1	Channel 2
00	no change	no change
01	no change	XOR LSB
10	XOR LSB	no change
11	XOR LSB	XOR LSB

Table 2.1: Pixel indicator channel relationship

from the message. We make use of the RGB color space in Chapter 3 without the use of an indicator channel.

2.4 FFmpeg

FFmpeg is an open source software suite of libraries for handling video, audio, and other multimedia files. Work from the software is used in programs such as Google Chrome, Davinci Resolve, VLC, and more. Because FFmpeg is open source, it is not unusual for the program to be modified and and published as an improvement to the original. Steganography can be used on open source software like FFmpeg because the hash signature of the original project would not be expected to be the same for a modified version of the project.

FFmpeg has multiple libraries with several files and header files. Program execution flow can vary greatly due to the many options and functions FFmpeg has for video, image, and audio manipulation and editing.

Among its multiple libraries, FFmpeg has an `avfilter` library that has options to alter decoded audio or video with a chain of filters. In particular, the `geq` filter, or "general equation change editor filter", can change a pixel's color component. A specified range of frames can be provided as input to apply the change to the pixel.

2.5 Python Image Library

To avoid the complication of making a filetype-specific solution to data hiding, an image processing library called `Pillow` is used. Pillow is a fork of the Python Image Library. Pillow focuses on image manipulation and editing. It is a relatively small library compared to FFmpeg. Pillow presents an interface to modify the pixel values of

an image. Using Pillow, a 2-D matrix of the image's pixels is generated. Each pixel in the 2-D matrix has a tuple of three RGB integers represented in decimal with a range of [0, 256[[10].

2.6 OpenCV

Similar to Pillow, OpenCV provides an interface to modify the pixels in a video frame. OpenCV uses FFmpeg's libraries for its backend and aided in reusing the steganographic methods used for our implementation [3].

Chapter 3

Dynamic steganography

3.1 An approximate idea

We propose that dynamic steganography is steganography in which the payload and cover are processes. The execution of the payload's process exists within the run-time of the cover process.

3.1.1 Components of dynamic steganography

Much like steganography in general, dynamic steganography utilizes the concepts of payloads and covers. For dynamic steganography, a *payload process*, *cover container* and *cover process* are required. The payload process is the process that will be executed in secret to external parties. The cover container can be a file, data directory, or software package in which the text representation of the payload process is embedded. The cover container is the medium to hide the instructions of the payload process. The cover process is a process that hides the payload process during execution. When the payload process is extracted and executed, it becomes a subprocess to the cover process. The payload process must start and end within the execution duration of the cover process. The pair of processes is used to accomplish the *dynamic* component of dynamic steganography, which is to hide a process within a process.

3.1.2 Cover container and cover process

The cover container and the cover process must be large and complex enough to hide the text representation of the payload process and the execution of the payload process respectively. The covers should be large enough to prevent suspicion. Suspicion is

difficult to quantify so an upper bound is defined as the real number ϵ, representing the proportion of the size of the payload to the size of its corresponding cover. The upper bound of ϵ can be made more strict depending on the implementation (typically $0 < \epsilon < 0.01$). The run-time of the cover process should be significantly longer than the run-time duration of the payload process.

3.1.3 Possible implementations

Dynamic steganography might benefit from the use of *self-modifying programs* to execute the payload process in parts. These parts are hereby referred to as *payload components*. The payload components make up the payload process. Their purpose is to execute parts of the payload process in iterative steps. For example, the iterative steps for video playback would be loading frame data from a video, the cover container. The cover process could modify its code to incorporate and execute the payload components during the cover process run-time.

The use of self-modifying programs could also benefit the security of the program from steganalysts by hiding the instructions in the cover process that pertain to dynamic steganography. The instructions that extract the payload components from the cover process would be written into the cover process during run-time and removed after the last payload component is completed. The instructions would only be part of the cover process during execution of the cover process.

Parallel computing might also benefit dynamic steganography. Payload components could be executed in parallel with the cover process. An additional thread to execute the payload components could reduce total run-time significantly. The cover process would serve as the main thread. The payload components would be retrieved from the cover container in iterative steps. A new thread would spawn to execute a payload process retrieved in a step, parallel with the cover process.

3.2 Uses

One use case of dynamic steganography is limited to a local machine, in which only the user will be aware of the steganographic operation being performed on the video for hiding or retrieving a program to execute. Should the user be monitored by external parties, the user could use dynamic steganography to perform tasks in secret to these parties.

Another use case is for one user to doctor a video with a payload process and send the video to a target recipient. The recipient views the video in order to retrieve the payload and execute the instructions without being detected by an external party. The external party may have access to monitor the hardware's display, the hardware performance, and the file system.

3.3 Attacking steganography

The stakeholders involved in dynamic steganography are the steganographer and the steganalysts. The steganographer intends to attack a computer system to execute a payload process within the execution of a cover process. The cover process performs steganographic operations on the cover container to retrieve instructions and execute the retrieved instructions. On the other hand, the steganalysts intend to detect the steganography. The steganalysts intend to attack the payload process. They may observe suspicious patterns in the cover container or suspicious activity in the cover process. Should the steganalysts have prior knowledge of the cover process, payload process, or cover container, they have options available to them to lead to increased suspicion to detect steganography. Figure 3.1 details the attack options for steganalysis for known components of dynamic steganography.

3.4 Data compression

Data compression is a technique that reduces the number of bits to represent the data. *Lossy compression* is a form of data compression which utilizes a quantizer; the quantizer reduces psycho-visual redundancies in the original image or video in an irreversible

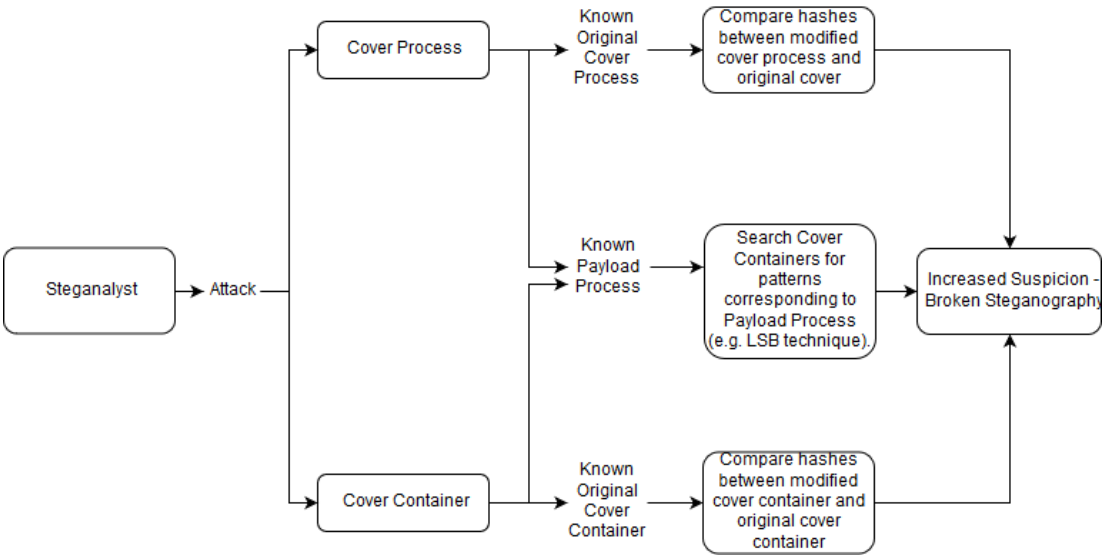

Figure 3.1: Attack model

way. The psycho-visual redundancies take advantage of the human visual system's inability to distinguish changes in an image. The JPEG image file type utilizes lossy compression. Issues were encountered when using JPEG for the implementation of image steganography in this thesis.

Lossless compression omits the quantizer in exchange for taking up more space in memory. Unlike lossy compression, lossless compression can reconstruct the original image from the compressed data. The PNG image file type utilizes lossless compression, which was used for the implementation of image steganography in this thesis. [11, 18]. Though there are algorithms to handle lossy compression, we opt to use lossless compression to avoid the complications of data loss in lossy compression [15, 18].

3.5 Modifying FFmpeg

FFmpeg has a library called `libavformat` which handles file I/O for media files as well as having other features for media manipulation. The file named `file.c` has function calls that are used by any operation performed with FFmpeg. To name a few, these operations include filetype conversions, video playback, and video filtering. The function calls the the `file_read()` and `file_write()` functions are made routinely when converting a video from one file type to another. The aforementioned functions

would be appropriate for applying the steganographic operations on the video frames.

The execution flow of FFmpeg is not clear. The command-line interface (CLI) of FFmpeg is well-documented; however, the source code does not have the same level of detail in its documentation for various functions. FFmpeg is managed by several people, with a multitude of contributors. The complexity and size of the software is a challenge. Filenames pertaining to encoding, decoding methods, or *codecs*, did not flush output from `printf()` to `stdout` when using the corresponding file types as input for FFmpeg. For example, an MP4 file does not appear to call functions in `mpeg.c`.

The fundamental data structure used in FFmpeg appeared to be `AVPacket` and `AVFrame`. `AVPacket` holds compressed data and its counterpart, `AVFrame`, holds uncompressed data. The challenges posed thereafter were to interpret the data that represented individual pixel color components and preserve the modified data during compression [12]. `AVFrame` is likely the best way to access the pixel data.

3.6 MP4 video format

Video containers are complex structures with required and optional data blocks. Taking a look at MP4 videos, pixel data are stored within the data stream of the video file in an `mdat` data block or *atom*. The initial atom, `ftyp`, contains information about the file type and version of the MP4 format, where the file type could be MP4 video or MP4 audio or some other variation. A separate atom, `moov`, is used to provide information about the mapping of the frames in the video [6]. Parsing the map file to modify the frames and their corresponding pixels in binary or hexadecimal proved to be a challenge. Additionally, the MP4 video format features lossy compression, indicating that some data would be lost in order to reduce the file size. A different video container like AVI uncompressed would be better suited for steganographic operations to prevent loss of data which could tamper with the payload process or payload in general.

Chapter 4

An approximate implementation of dynamic steganography

4.1 Image steganography

We propose that a video frame is similar to an image. We implement image steganography as a base for video steganography detailed The PNG file format for images utilizes lossless compression, presenting itself as a suitable candidate for image steganography. We use the Pillow library to avoid parsing the organized data in PNG known as `chunks` [4]. This implementation modifies one of the RGB color components in a pixel, cycling between red, green, and blue. Pixels are chosen by a function that is used for embedding and retrieving the message.

4.2 Choosing pixels

The next pixel chosen for hiding information is given by the following formula:

$$\lfloor x + b + (a(i+j)^r \bmod p \rfloor)$$

The function `getNextPixel(.)` takes the position of the current pixel from a one dimensional representation of the pixel array. The variable x represents the position of the current pixel. A minimum offset b determines the shortest possible distance the next pixel can be from the current pixel in one dimension. The a in the expression is a multiplier variable that can be adjusted. The i is the index of the current character in the message to be hidden, and j is the index of the current bit with respect to the character. The exponent r is a constant with a default value of 2 for the quadratic residue function, but it is possible to use other values for the exponent. p is a prime number which determines the maximum spacing between the pixels to be $b + (p - 1)$.

The function `getNextPixel(.)` computes quadratic residues (i.e. $r = 2$) to mimic a pseudo-random generator to produce values to obtain the positions of all pixels used to hide the bits of a message. Quadratic residues approximately follow a Poisson distribution for high values of p, where p is the prime number in the modulo operation.

q is a quadratic residue if there exists an integer x such that:

$$x^2 \equiv q \bmod p$$

For small values of p, a prime number, the quadratic residues can result in palindromic patterns [20]. A pattern can present an issue for steganography as it may serve as an antecedent to focused suspicion, reinforcing a belief that the examined material may be tampered.

This presents a difficulty for the implementation as we try to maximize p, which determines the maximum spacing between pixels in `getNextPixel(.)`. If the spacing between the pixel selection is too large, then the size of the message will be limited. If multiple color components are changed in a pixel instead of one per pixel, then more data can be hidden; however, that would also change the color composition in a particular pixel more and adds the risk of suspicion. Large spacing could be accommodated in a video file which would increase the capacity by the amount of frames within the movie. Today, films are projected at a minimum of 24 frames per second with a resolution of 1920 by 1080 [8]. Assuming a video lasts 120 minutes, and the frame rate is set at 24 frames per second, the capacity of the video would increase by a factor of 172800. Should the quadratic residue method not be as effective as an alternative, the function `getNextPixel(.)` could be replaced by another function that returns a value for the position of the next pixel.

4.3 Hiding the message

The *embed method* detailed in Algorithm 1 hides the message payload into the cover. In this case, the message payload is a string of ASCII characters, and the cover is an image. The LSB technique is used to hide the bits of the message payload in the color components of the pixels.

A 2-D array of pixel coordinates is generated for the loaded image using Python Image Library. The initial values for the X and Y pixel coordinate begin at (0, 0). The position of the next pixel is determined from the function `getNextPixel(.)` from a 1-D array representation. The position of the pixel in a 1-D array is converted into coordinates of the pixel in the 2-D array. A color is obtained with the `getColorIndex(.)` function. A square residue function with modulo 3 is used to rotate through the color components of the pixel, red, green, or blue.

Prior to hiding the message in the image, we hide a total of 28 bits of *metadata* into the image. The metadata contains information about the location of the payload. Four bits are stored in the image representing a multiplier value used in the function `getNextPixel(.)` and an additional four bits are stored for a minimum offset value, also used in the same function. The multiplier and offset variables are fixed for the embedding of this metadata; however, the pixels selected to hide the message will use the values provided by these metadata values. The remaining 20 bits are to store the length of the hidden message, represented as d, where $d \leq 2^{20}$.

Incorporating the message length indicates how many extended ASCII characters must be read to retrieve the message from the image. If the length is not communicated for use in the retrieval method, then there could be some extraneous bits that would be translated into ASCII characters that may affect the meaning of the message or present itself as some unrecognizable text.

The message payload is converted from extended ASCII character representation to a 2-D array of bits, with each row of bits corresponding to an extended ASCII character, supporting up to 256 characters. The implementation is constrained to using one type of character encoding to convert the message to binary and vice-versa.

We iterate over each bit of the message to be stored and compare its value to the value of the LSB of the chosen color component in the selected pixel. If the bit values match, no action is taken; however, if the values are different, `XOR` is applied to the LSB of the selected color component and selected bit from the payload message. Each pixel selected by the function `getNextPixel(.)` carries a bit from the message in one of its color components.

Algorithm 1 Embed method

```
1: procedure EMBED_MESSAGE(filename, a, b, p)
2:     # Let d be the length of the message payload
3:     # Let a be the multiplier represented in binary
4:     # Let b be the minimum offset of pixel to use, represented in binary
5:     # Let c be the index of the color component
6:     # Let h be a 28 bit string concatenation of a (4 bits), b, (4 bits), and d (20 bits)
7:     # Let A be a 2-D matrix to represent the pixel map of the image
8:     # Let m be the number of rows in A
9:     # Let n be the number of columns in A
10:    # Let M be a 2-D array to represent the bits of ASCII chars
11:
12:    int d, a, b, c, m, n, X, Y absPosition
13:    int Array A[., .], M[., .]
14:    A ← pixel map of the loaded image via PIL                          ▷ A[m, n]
15:    M ←message_to_binary(message)    ▷ M[d, 8] where 8 is the number of bits for
16:                                                          ▷ extended ASCII character
17:    for j ← 0, 28 do
18:        absPosition ← getNextPixel(n ∗ Y + X, m, n, 1, 1, 0, j)
19:        X ← absPosition% n
20:        Y ← absPosition// n
21:        c ← getColorIndex(j)
22:        if bin(A[X, Y][c])[−1] ≠ h[j] : then
23:            if c == 0 then
24:                A[X, Y] ← (A[X, Y][0] ⊕ 1, A[X, Y][1], A[X, Y][2])      ▷ ⊕ is bitwise
   XOR
25:            else if c == 1 then
26:                A[X, Y] ← (A[X, Y][0], A[X, Y][1] ⊕ 1, A[X, Y][2])
27:            else if c == 2 then
28:                A[X, Y] ← (A[X, Y][0], A[X, Y][1], A[X, Y][2] ⊕ 1)
29:    for i ← 1, d + 1 do
30:        for j ← 0, 8 do
31:            absPosition ← getNextPixel(m ∗ Y + X, m, n, a, b, 8 ∗ i, j)
32:            X ← absPosition% n
33:            Y ← absPosition// n
34:            c ← getColorIndex(j)
35:            if bin(A[X, Y][c])[−1] ≠ h[j] : then
36:                if c == 0 then
37:                    A[X, Y] ← (A[X, Y][0] ⊕ 1, A[X, Y][1], A[X, Y][2])
38:                else if c == 1 then
39:                    A[X, Y] ← (A[X, Y][0], A[X, Y][1] ⊕ 1, A[X, Y][2])
40:                else if c == 2 then
41:                    A[X, Y] ← (A[X, Y][0], A[X, Y][1], A[X, Y][2] ⊕ 1)
```

4.4 Retrieving the message

Once the message has been hidden into the image, a retrieval method must also exist to read the message from the image. The method `getNextPixel(.)` determines which pixels were used for embedding and the retrieval method will obtain the bits of information from the image in order to reassemble the message.

Algorithm 2 Retrieval method

```
1:  procedure RETRIEVE_MESSAGE(filename)
2:      # Let a be the multiplier represented in binary
3:      # Let b be the minimum offset of pixel to use represented in binary
4:      # Let c be the index of the color component
5:      # Let h be an empty 1-D array of a, b, and d
6:      # Let A be a 2-D matrix to represent the pixel map of the image
7:      # Let m be the number of rows in A
8:      # Let n be the number of columns in A
9:      # Let M be an empty array to store arrays of bits of payload message
10:
11:     int d, a, b, c, m, n, X, Y, absPosition
12:     int Array A[.,.], M[.,.] B[.], h[.]
13:     A ← pixel map of the loaded image via PIL                    ▷ A[m, n]
14:     M ←message_to_binary(message)                                ▷ M[d, l]
15:     for j ← 0, 28 do
16:         absPosition ← getNextPixel(n ∗ Y + X, m, n, 1, 1, 0, j)
17:         X ← absPosition % n
18:         Y ← absPosition // n
19:         c ← getColorIndex(j)
20:         h ← append(bin(A[X, Y][c])[-1]
21:     a ← int(h[0,4])
22:     b ← int(h[4,8])
23:     d ← int(h[8,28])
24:     M ← [d, 8]
25:     for i ← 1, d + 1 do
26:         for j ← 0, 8 do
27:                                          ▷ 8 number of bits for extended ASCII
28:             absPosition ← getNextPixel(m ∗ Y + X, m, n, a, b, 8 ∗ i, j)
29:             X ← absPosition% n
30:             Y ← absPosition// n
31:             c ← getColorIndex(j)
32:         M[i − 1][j] ← bin(A[pixelX, pixelY][c])[−1]
```

The *retrieval method* detailed in Algorithm 2 processes the first twenty-eight pixels with the function `getNextPixel(.)`, supplied with fixed multiplier a and a fixed offset

b. Retrieving these bits provides the values of a and b for the pixels carrying the hidden message as well as the length of the message to read d characters.

As previously mentioned, extended ASCII supports 256 characters. Each character is represented by eight bits and there are d characters. In total, there are $8d$ bits to retrieve. `getNextPixel(.)` and `getColorIndex(.)` are used to locate the pixels and color components used for the embed method. The LSBs in these locations are stored in a 2-D array. The array is used as an argument to a `binaryToMessage()` function that transforms the bitstrings into their corresponding extended ASCII representations. The end result of the retrieval method is the original message hidden in the designated pixels.

4.5 Animated image steganography

We attempted to convert the images from PNG to GIF format. GIF supports animated images. Prior to attempting to produce the animated image from a set of images, we converted a single PNG image to GIF format. We discovered that while both formats use lossless compression, the GIF had reduced quality. Our PNG image in 5.7a has a color representation of 24 bits per pixel. GIF represents color in 8 bits per pixel. An image in a GIF can reference 256 colors from a 24 bit RGB color space. While it is possible to change the color palette to better represent the diversity of colors in the image, colors remain extremely limited in GIF [7]. This limitation could distort the payload message. In dynamic steganography, the payload message would be replaced by a payload process; the distortion of the payload process could result in errors and undefined behavior, risking exposure to the steganalyst.

4.6 Video steganography

We propose that video steganography can be perceived as an extension of image steganography, in which a video is a sequence of images [15]. Algorithm 1 and 2 were used with OpenCV library which interfaces with FFmpeg as a backend for video editing and manipulation. Similar to Pillow's image manipulation, OpenCV has functionality

to modify frames and their corresponding pixels.

Chapter 5

Discussion

5.1 Image and video steganography results

Multiple images were tested with the embed and retrieval methods. The difference between the original images and the doctored images were imperceptible to the human eye.

(a) Original image (b) Altered image

Figure 5.1: Test image comparison

Figure 5.7a is the original image and Figure 5.7b is modified image, in which the embed method is applied. The payload message used in this case is 1,050 characters of Lorem Ipsum. The change from the original image to the new image is imperceptible to the human eye.

Figure 5.2 shows RGB histograms representing Figure 5.7a and Figure 5.7b respectively. Again, there is a minute difference between the two images. RGB histograms were generated for multiple images yielding similar results. The cover image is large enough, with respect to pixel count, to obfuscate the modified pixels. The LSB technique makes the smallest change to the pixel's chosen color component. This makes it difficult for the human eye to notice the alterations made to the image.

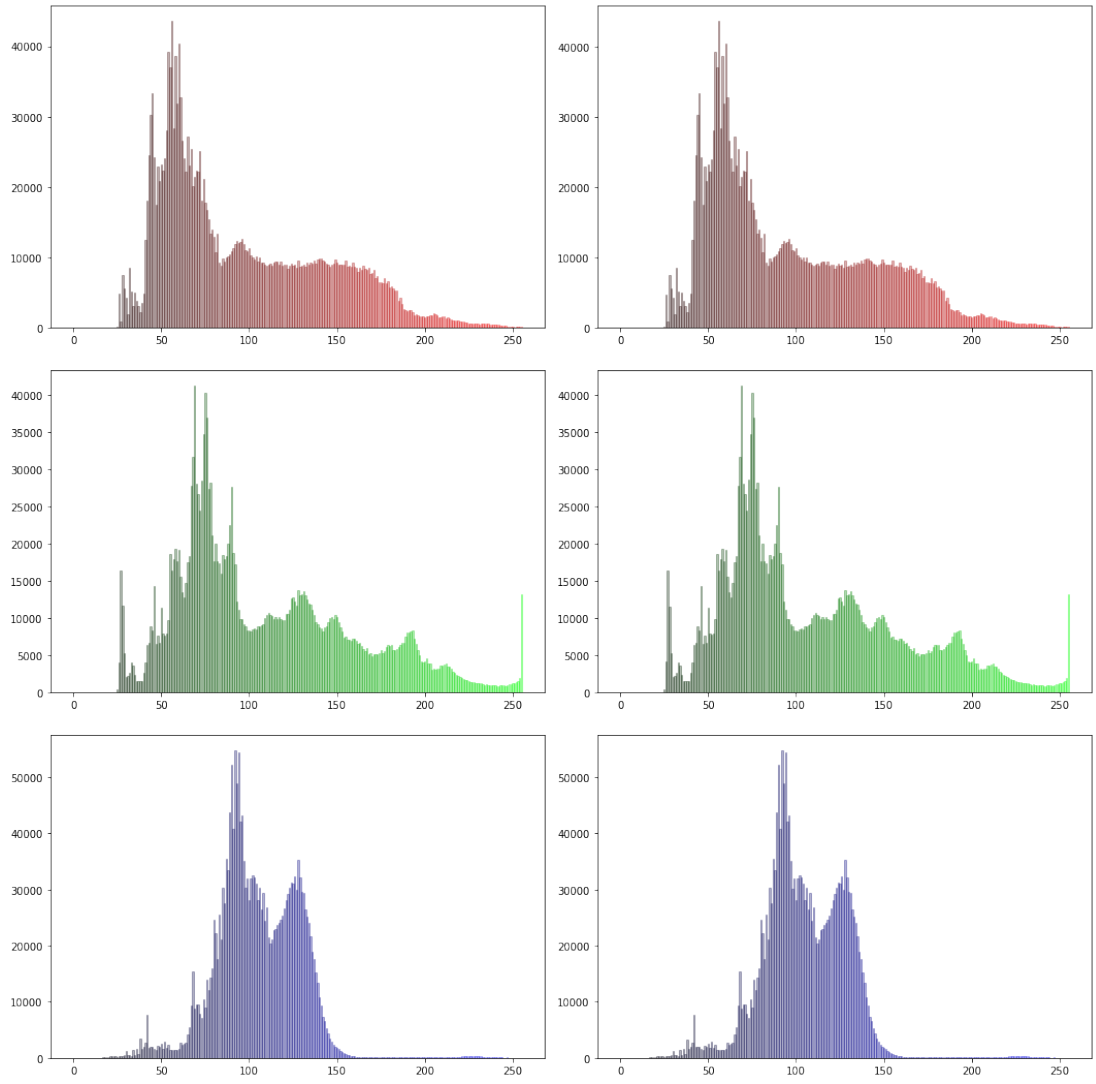

Figure 5.2: RGB histograms

Distinguishing the changes become apparent when computing the differences between the histogram data for the original image and the modified image in Figure 5.7. In Figure 5.3, the data shows the amount of additions or subtractions. Taking the sum of the values in each color component yields zero, suggesting that for every color removed from the original image, a new color was added. The additional color is different than its counterpart by 1 in decimal, which follows with changing the LSB in one color component.

To further distinguish the modifications to an image, we start with the original image, highlight affected pixels, and show the final result for a solid white image in

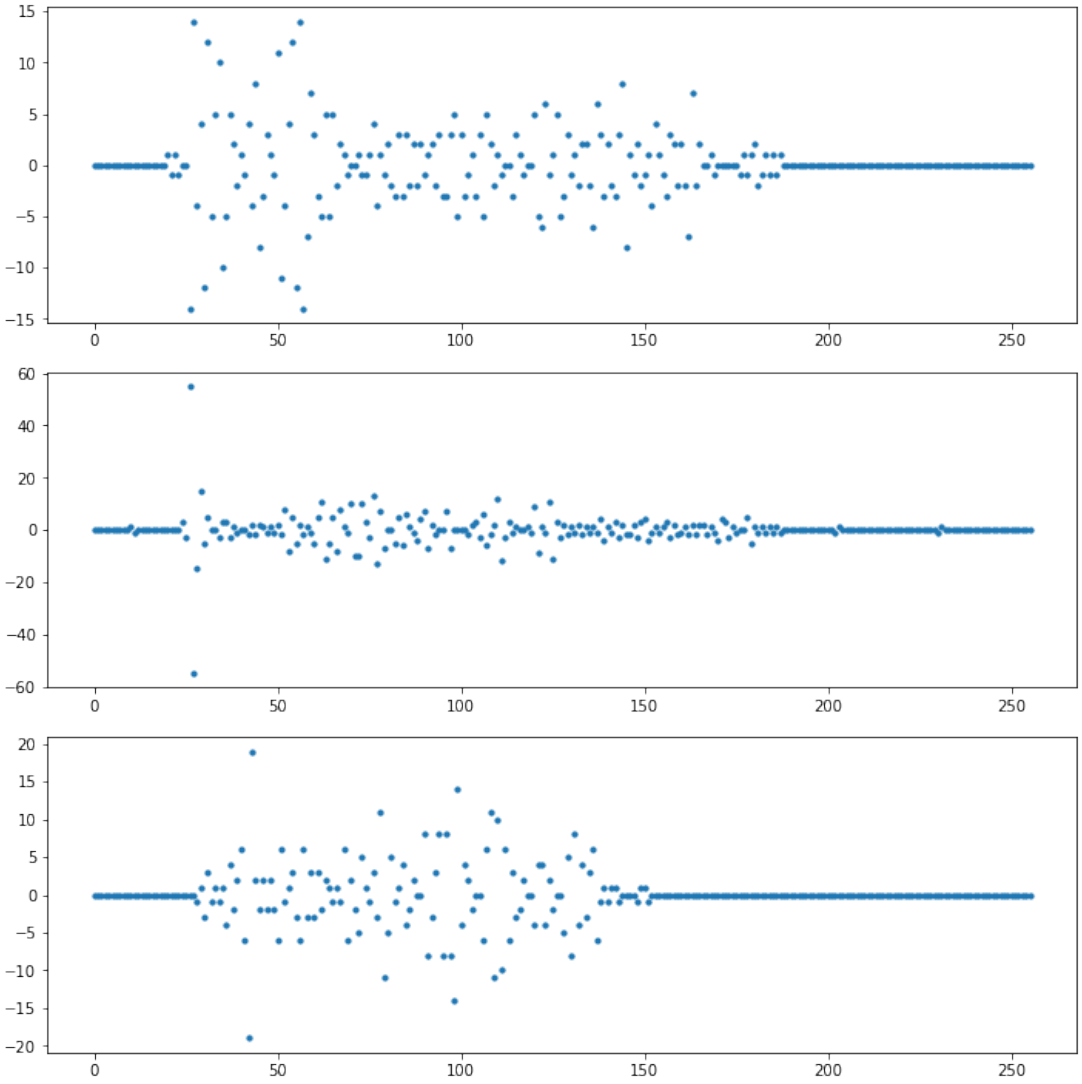

Figure 5.3: Difference of color value counts

Figures 5.4-5.6. Figure 5.4 is a 1920 x 1080 image. Each pixel of the image is white or 0xFFFFFF in hexadecimal. The payload process is 1,050 characters of Lorem Ipsum. In Figure 5.5, we highlight the changes made to the image by changing the color of the pixel to the color of the chosen color component, with no contribution from the other color components. For example, if the red color component was chosen for a given pixel, then the pixel's color would be changed to (255, 0, 0) RGB values in decimal.

The pixels are clearly highlighted in Figure 5.5. The modified pixels cover the top portion of the image. If the message was longer, it would take up more space in the image. Having the modified pixels close to each other may arouse suspicion [13].

Though suspicion cannot be quantified at this time, it may be argued that concentrating hidden data in a particular location may increase the likelihood of detection. In response to this, the parameters b and p in `getNextPixel(.)` can take on different values to increase the minimum and maximum spacing of the selected pixels. The closeness of the pixels may not necessarily be a bad thing. Should the image be manipulated by an image editor, most of the image can be modified without affecting the hidden data.

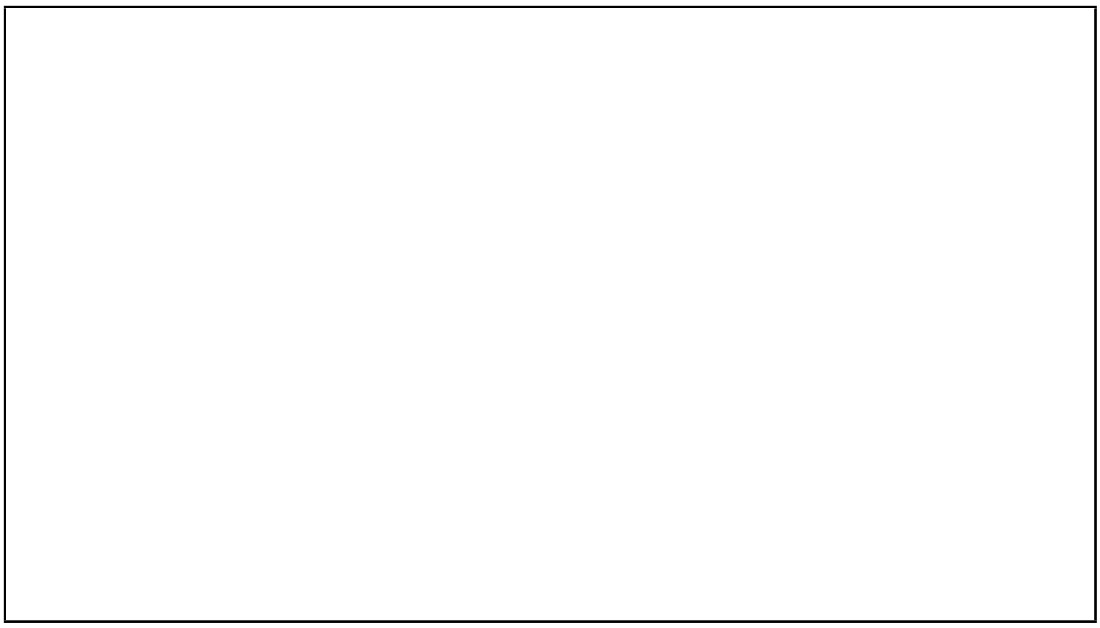

Figure 5.4: Original white image

Despite the hidden data being clustered together, it remains to be difficult to perceive without the highlighting performed on Figure 5.5. Figure 5.6 displays the results of the embed method when applied to the image in Figure 5.4.

The process to highlight the pixels modified in an image composed entirely in one color is feasible and simple to check. Opening the image in an image editor such as `mspaint.exe`, selecting the "fill with color" option and subsequently filling with black will change all the white pixels with (255, 255, 255) in RGB value to black, represented as (0, 0, 0). By doing this, the pixels that have a 254 in any one of the color components will be revealed (i.e. (254, 255, 255) (255, 254, 255) or (255, 255, 254)). These colors exist due to the `XOR` applied to the color value to represent the payload process. While this is a simple method to reveal the pixels used to hide the data,

the role of steganography is to prevent detection. Cryptography is the complement to steganography and is effective in securing the data once the message payload is detected.

We found success in hiding the payload process in a video, but ran into difficulty retrieving the payload process after the output video is compressed and saved. Lossy compression method, `DIVX`, and lossless compression method, `FFV1`, were used. `FFV1` was better for use; however, the colors of all the pixels in the frames varied slightly compared to the frames prior to compression and saving to the output video.

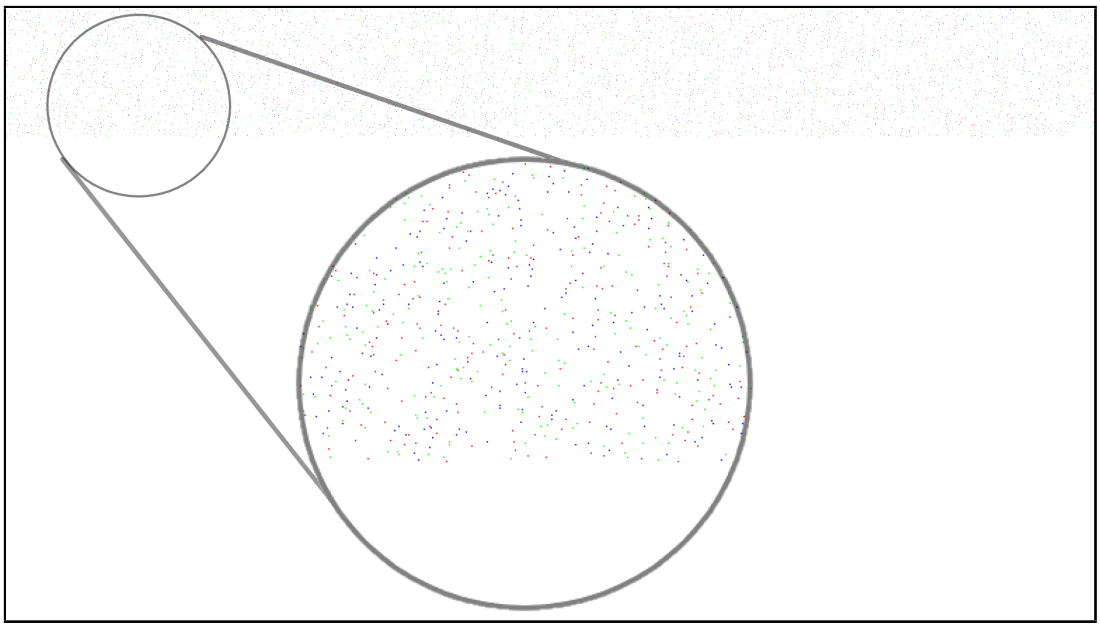

Figure 5.5: Highlighted white image

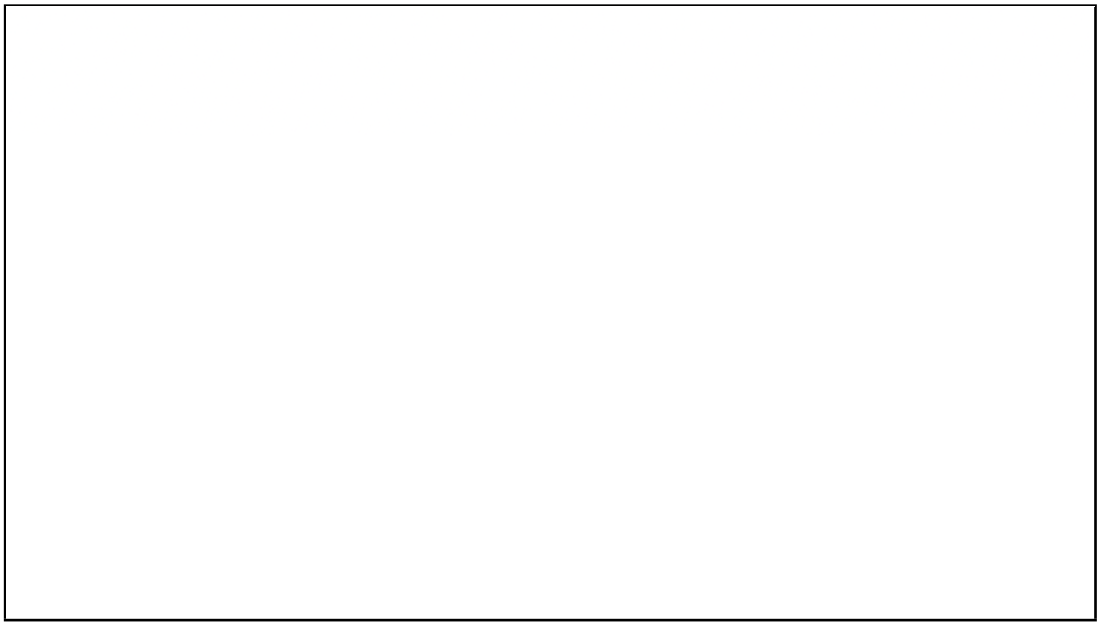

Figure 5.6: Modified white image

(a) Original image (b) GIF image

Figure 5.7: PNG and GIF image comparison

Chapter 6

Conclusions and future work

6.1 Impediments

6.1.1 MP4

Video steganography attempts began with the MP4 video container. The container is frequently used in social media due to its lossy compression method, which is efficient at reducing the size of the video. The space-saving efficiency can conflict with steganography by changing the payload.

The MP4 video file structure is complex to manipulate directly in bits. The organization of the data for the file format is inconsistent. The blocks of data known as atoms are not always arranged in the same order. This property of the format presented constraints for one steganography implementation which hides a `TrueCrypt` container. The `TrueCrypt` container is a block of data that is steganized into a file. The implementation creates a `TrueCrypt` container with the payload of hidden data and some dummy data and inserts the `TrueCrypt` container into the video file in a place where there are free bytes available before the `mdat` atom when the atoms are arranged in `ftyp-moov-mdat` order. The implementation is constrained by particular order of the video file. The length of certain atoms, are also constrained depending on the order of the atoms. The hidden data are not displayed in the playback of the video, because they are not a part of the video datastream, the mdat atom [2].

6.1.2 Compression methods

Additionally, the compression used for the MP4 file is lossy. As mentioned earlier, lossy compression uses a quantizer that removes psycho-visual redundancies. These

redundancies may be a part of the payload process. Any loss of data from the payload process could break the process and possibly expose itself to the steganalyst. The payload process could also result in undefined behavior and could produce undesired results.

To resolve the aforementioned issue, implementations of lossy compression algorithms would have to be investigated further to apply steganographic techniques to widely used compression methods. The payload process would have to be hidden in the cover container at positions that do not meet criteria for compression or after the quantization phase where data are reduced. The payload process could be embedded after compression and before decompression to avoid data loss. [11].

6.1.3 Pixel format

We were able to hide a payload process in a cover container but unable to retrieve the payload component successfully using the `FFV1` codec. Values were correct after modifying the frame and incorrect when writing the frames to the video, compressing, and saving the frames using `FFV1`, an open-source lossless codec by FFmpeg. The culprit could be the `pixel format` which determines the color space representation for the pixel. We focused on RGB; however, certain video containers may support a small set of pixel formats. `FFV1` supports multiple but colors do not appear to save accurately. This inaccuracy may be attributed to rounding errors between different types of colorspaces (e.g. RGB or YUV). `FFV1` is the one of the few lossless compression methods that worked for the local machine; however, we also tried installing the Lagarith codec which also supports lossless compression. With Lagarith, we were able to successfully hide data inside the video. Each frame hides part of the payload process. Upon retrieval, each part is executed by Python's builtin `exec(.)` function. The function does not support `return` to the parent scope; however, it does inherit from the calling scope. This approximate implementation for dynamic steganography shows the possibility but does not yet embrace its full potential. Referring to Chapter 3.1.3, self-modifying programs or parallel processing programs could achieve more than `exec(.)` . [1, 3].

6.2 Research paths

6.2.1 Sharpening the idea of dynamic steganography

We introduced the idea of steganography. The concept is in its infancy and will benefit from added discussion. We propose that dynamic steganography is an extension of traditional steganography, which is static in comparison. Dynamic steganography could potentially be applied to computationally expensive programs, such as machine learning software and numerical methods software.

6.2.2 Watermarks

Fulfilling the attempt at implementing video steganography could also be used to further research in the field of digital watermarks. Should the payload process not be divided into payload components, the payload process could be written over multiple frames.

6.2.3 Viruses

The concept of dynamic steganography is very similar to the concept of the trojan virus mentioned earlier. By gathering more knowledge on how viruses bypass the operating system, dynamic steganography could be improved by hiding the payload components on a target machine.

6.2.4 Disabling steganography and steganalysis

In learning how attacks can be performed, research in steganalysis to combat dynamic steganography can develop. In the instance of dynamic steganography using videos, it is difficult to analyze every video; however, if the frames contain payload components, determining a pattern in a frame is enough to arouse focused suspicion. Videos that exhibit these patterns in one frame can be quarantined for further investigation.

Interfering with steganography may aid in disabling steganography. For image, video, and perhaps dynamic steganography, transforming the cover slightly can interrupt the payload. An example of transformation is applying a color filter to the cover. This could also be thought of as offseting the three color values in each pixel.

Some transformations may be too expensive. Steganalysts would benefit from research looking into the efficiency of image or video transformations to disable steganography. Using arithmetic to change all the LSBs in each color component to zeroes or ones. The colors could also be increased or decreased by some specified value to distort the steganography and possibly break it.

References

[1] Ffv1 video coding format version 0, 1, and 3. URL https://tools.ietf.org/html/draft-ietf-cellar-ffv1-05. (Visited on 12/24/18).

[2] Real steganography with truecrypt. URL https://keyj.emphy.de/real-steganography-with-truecrypt/. (Visited on 10/11/18).

[3] Getting started with videos. URL https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_gui/py_video_display/py_video_display.html. (Visited on 12/23/18).

[4] Png (portable network graphics) specification, version 1.2. URL http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html. (Visited On 5/19/18).

[5] Raster & vector. URL https://helpx.adobe.com/photoshop-elements/key-concepts/raster-vector.html. (Visited on 9/2/18).

[6] URL https://www.klennet.com/carver/video-recovery.aspx. (Visited on 8/20/18).

[7] Gif, Dec 2018. URL https://en.wikipedia.org/wiki/GIF. (Visited on 11/25/18).

[8] Ntsc, Dec 2018. URL https://en.wikipedia.org/wiki/NTSC. (Visited on 11/1/18).

[9] Raster images vs. vector graphics, Jun 2018. URL https://www.printcnx.com/resources-and-support/addiational-resources/raster-images-vs-vector-graphics/. (Visited on 4/15/18).

[10] A. Clark. Pillow. URL https://pillow.readthedocs.io/. (Visited on 9/2/18).

[11] F. Ernawan, N. A. Abu, and N. Suryana. Tmt quantization table generation based on psychovisual threshold for image compression. In *2013 International Conference of Information and Communication Technology (ICoICT)*, pages 202–207, March 2013. doi: 10.1109/ICoICT.2013.6574574.

[12] FFmpeg. Ffmpeg/ffmpeg, Dec 2018. URL https://github.com/FFmpeg/FFmpeg. (Visited on 3/13/18).

[13] A. A.-A. Gutub. Pixel indicator technique for rgb image steganography. *Journal of Emerging Technologies in Web Intelligence*, 2(1), Mar 2010. doi: 10.4304/jetwi.2.1.56-64.

[14] T. Jamil. Steganography: the art of hiding information in plain sight. *IEEE Potentials*, 18(1):10–12, 1999. doi: 10.1109/45.747237.

[15] M. M. Sadek, A. S. Khalifa, and M. G. M. Mostafa. Video steganography: a comprehensive review. *SpringerLink*, Mar 2014. URL https://link.springer.com/article/10.1007/s11042-014-1952-z.

[16] D. Salomon. *Elements of computer security.* Springer, 2010.

[17] Y. Schul, E. Burnstein, and A. Bardi. Dealing with deceptions that are difficult to detect: Encoding and judgment as a function of preparing to receive invalid information. *Journal of Experimental Social Psychology*, 32(11):228–253, 1996.

[18] V. Singh, O. P. Singh, and G. R. Mishra. A brief introduction on image compression techniques and standards. *International Journal of Technology and Research Advances*, 2013(2):15–21, 2013.

[19] K. Thangadurai and G. S. Devi. An analysis of lsb based image steganography techniques. *2014 International Conference on Computer Communication and Informatics*, 2014. doi: 10.1109/iccci.2014.6921751.

[20] S. Wright. *Quadratic residues and non-residues: selected topics.* Springer, 2016.