Spring 3-2021

JRevealPEG: A Semi-Blind JPEG Steganalysis Tool Targeting Current Open-Source Embedding Programs

Charles A. Badami
Dakota State University

JREVEALPEG: A SEMI-BLIND JPEG STEGANALYSIS TOOL TARGETING CURRENT OPEN-SOURCE EMBEDDING PROGRAMS

A dissertation submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Cyber Operations

March 2021

By
Charles A. Badami

Dissertation Committee:

Dr. Wayne Pauli
Dr. Cody Welu
Dr. Christopher Olson

DISSERTATION APPROVAL FORM

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Charles Badami

Dissertation Title: JRevealPEG: a semi-blind JPEG steganalysis tool targeting current open-source embedding programs

Dissertation Chair/Co-Chair: Wayne Pauli Date: April 4, 2021

Name: Wayne Pauli

Dissertation Chair/Co-Chair: _____ Date: _____

Name:

Committee member: Cody Welu Date: April 4, 2021

Name: Cody Welu

Committee member: Christopher Olson Date: April 5, 2021

Name: Christopher Olson

Committee member: _____ Date: _____

Name:

Committee member: _____ Date: _____

Name:

Original to Office of Graduate Studies and Research
Acid-free copies with written reports to library

ACKNOWLEDGMENT

I would like to express my deepest appreciation to my doctoral committee, to all of my professors and mentors at DSU, and to my friends and colleagues at Northwest Missouri State University for all of your help and support throughout this process. I especially want to thank my wife and children for putting up with me and tolerating my frequent disappearances while I obsessed over this project.

ABSTRACT

Steganography in computer science refers to the hiding of messages or data within other messages or data; the detection of these hidden messages is called steganalysis. Digital steganography can be used to hide any type of file or data, including text, images, audio, and video inside other text, image, audio, or video data. While steganography can be used to legitimately hide data for non-malicious purposes, it is also frequently used in a malicious manner. This paper proposes JRevealPEG, a software tool written in Python that will aid in the detection of steganography in JPEG images with respect to identifying a targeted set of open-source embedding tools. It is hoped that JRevealPEG will assist in furthering the research into effective steganalysis techniques, to ultimately help identify the source of hidden and possibly sensitive or malicious messages, as well as contribute to efforts at thwarting the activities of bad actors.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

Charles A. Badami

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Steganography in computer science refers to the hiding of messages or data within other messages or data; the detection of these hidden messages is called steganalysis. Digital steganography can be used to hide any type of file or data, including text, images, audio, and video inside other text, image, audio, or video data. Zielinska, Mazurczyk, and Szczypiorski (2014) point out that people sometimes confuse steganography with cryptography. The main purpose of both is to ensure confidentiality of a message. They are distinguished, however, by what exactly is being hidden. In cryptography, the message itself is being obfuscated (meaning garbled, obscured, or made unclear) whether or not an observer knows it is being sent. In steganography, the fact that the message is being sent is hidden; it is the channel of communication that is being kept secret (Zielinska et al., 2014).

The practice of steganography in general has roots that go back millennia. Jamil (1999) describes several historical examples, some very ancient, and some from within the last century. There is the story of a nobleman in Medea hiding a message in the belly of an unskinned hare, which was delivered by someone dressed as a hunter. Another account involves a Persian tattooing a secret message on a slave's shaved head, waiting for the hair to grow back, then sending the slave to the recipient, at which point he was to shave his head and reveal the message. One commonly used historical technique is to write a message on some sort of paper medium in a kind of "invisible" ink, e.g. milk, or certain types of juices. The hidden message could then be revealed by subjecting the paper to heat. In more recent history, the Nazi spy George Dasch used copper sulfate on a handkerchief as invisible ink that only became visible when subjected to ammonia fumes. Another technique used by the Germans in World War II was to use objects called "microdots". A microdot was an extremely small photograph about the size of a period; the receiver could blow up the microdot to reveal a full page of information. Not to be outdone, the United States during WWII employed Navajo speakers as so-called "codetalkers" to send secret radio messages in their native tongue. Only 28 non-Navajos (none of them German or Japanese) were thought to

be able to speak the language at the time, and the codetalkers made it even more difficult by using slang. Even during the Gulf War in 1990 to 1991, some Navajos used a similar method to bypass radio censors and send messages to their loved ones serving overseas (Jamil, 1999).

Burney (2018) cites two more historical steganography examples of interest. One account, again from WWII, mentions the smuggling of Monopoly games into German prison camps. Maps, files, and compasses were hidden in the game which were intended to aid British prisoners in escaping. A second story involves a rumor regarding former British Prime Minister Margaret Thatcher, alleging that Thatcher arranged to have "word processors programmed to encode the identity of the writer in the word spacing," so that she could track cabinet ministers who may have been leaking documents to the press (Burney, 2018).

Steganography has been practiced throughout history, involving a variety of techniques, and its use cases have ranged from personal errands to wartime tactics. The next section discusses recent examples of the malicious use of steganography in the digital world, which is among the primary concerns of this study.

Problem Statement

While steganography can be used in the digital world to legitimately hide data for non-malicious purposes, it is also frequently used in a malicious manner. Burney (2018) notes that one common, legitimate purpose for steganography includes secretly marking a document to be able to trace its authenticity, in an effort to discourage stealing, unauthorized use, or plagiarism. However, the malicious use of steganography is becoming more prevalent and can have severe consequences. Shulmin and Krylova (2017) note the increasing use of steganography by those creating malware and by perpetrators of cyber-espionage, while also stating that most current anti-malware tools do not provide much protection. Vijayan (2017) reports that image steganography in particular is of primary concern, as image files are commonly used for command and control communications, for receiving and exfiltrating data. Current research continually adds to the list of known malware that uses steganography, some of which includes Powload, VeryMal, Novel, AdGholas, Fakem, and StegoLoader (Brunot, 2019). Examples of activity involving the use of this malware can be found relatively quickly through Internet searches. One source reports that Powload had a surge in usage in the first part of 2018, mainly distributed through email spam (Cisomag, 2019). Dunaway (2019)

details an attack by VeryMal in March of 2019 which was connected to Google's Firebase platform and lasted about two days, affecting about a million users. A third article reports that AdGholas was used to direct traffic to malicious advertising sites for over a year without being detected (mid-2015 to mid-2016), drawing one to five million hits each day (Kafeine, 2016).

Given that malicious image steganography is becoming more prevalent and that anti-malware tools are often inadequate to identify and protect against it, a clear need exists for new and ongoing research into updated steganography detection methods, with the primary goal of aiding security professionals and researchers in their efforts to mitigate this threat. To that end, this study details the design of JRevealPEG, a software tool that assists in the detection of steganography in JPEG images with respect to identifying a targeted set of open-source embedding tools. It is intended that this improvement in the detection of steganography and the tools used to create it will help researchers identify malicious content and prevent breaches before they occur, as well as help authorities trace such content to its origin and possibly expose the tools and adversaries responsible. This will, in turn, contribute to a consumer's overall safety in the digital realm and the progress of cyber security research in general.

As mentioned above, incidents of digital steganography in conjunction with malware and nefarious activities can be extremely pervasive, persistent, and reach a large number of victims in a short amount of time. The design of JRevealPEG is intended as a response to this problem, and the main purpose of this study is delineated in the following section.

Purpose of the Study

The purpose of this study was to help address the problem stated above by designing a software artifact that detects steganography in JPEG images while focusing on a select group of open-source tools. The artifact was programmed in the Python language, and a single-case mechanism experiment (discussed in Chapter 4) was used to validate the program. Several prepared JPEG samples were given to the program as input for the experiment under controlled conditions. Measurements of accuracy, timing, and file size were taken and used in analysis.

Design science was the methodology used for this study, as delineated by Weiringa (2014) and explained in Chapter 3. The observations and lessons learned from the design science process for this artifact, the results of the validation experiment, and certain statistical measurements were documented and analyzed. The results of this study are hoped to yield a direct contribution to steganalysis research and spawn practical applications for cyber security personnel in all types of organizations.

The purpose of the study is focused on the design of JRevealPEG as a steganalysis tool. The next section details the motivation behind this study in terms of what is lacking in other existing detection tools and methods.

Motivation

If the intent is to use digital steganography to embed and send hidden messages, files, and other data, a cursory Internet search reveals that many free and easy-to-use tools are readily available. Given the seriousness of the possible malicious use of these programs, some of the current research in this domain focuses on evaluating the effectiveness of existing methods that are meant to detect this steganography. A recent study by Serrano (2019) tested steganalysis tools, i.e. software that detects steganography. The study investigated several types of carrier files, including image, audio, and video files. In terms of image files, JPEGs were tested, as well as GIFs and PNGs. The detection tools tested included VSL, StegSecret, and StegDetect. Serrano found that the tools tested had two significant types of limitations: first, the number of successfully detected image carrier files (those with hidden data) was generally rather low, with an average detection rate of 3.75 images out of 15; second, several tools lacked or had minimal ability to identify the tool that embedded the steganography, resulting in an average identification rate of one tool out of six (Serrano, 2019). The artifact proposed in the present study, JRevealPEG, is intended to address both of these shortcomings of current steganalysis tools.

As described in the Literature Review, the vast majority of recent image steganography detection methods depend upon supervised machine learning to make their determinations. Supervised learning involves data that is classified by humans ahead of time and often requires thousands of samples to be used as a training set. In addition, Qiao, Luo, Wu, Xu, and Qian (2019) observe that there is a lack of modern research into steganalysis

methods that are "unsupervised," referring to machine analysis of data sets that are not labeled by humans beforehand. Unsupervised techniques find patterns based on the data points alone and do not require training sets. Although it does not use machine learning, JRevealPEG is nonetheless intended to help fill this gap by utilizing detection methods that have no need for training sets.

Additionally, JPEG compression is acknowledged to be the most popular standard among all image types, with several billion JPEGS created daily as of 2015 (JPEG, n.d.). This fact combined with the prevalence of JPEG steganography research shown in the Literature Review is a major reason why the proposed tool focuses on JPEG steganalysis.

As referenced above, existing image steganalysis tools can be shown to lack effectiveness when it comes to attribution and specific embedding tools. It is also noted that many detection methods rely on machine learning and large training data sets. This study focuses on JPEG analysis due to its immense popularity as an image standard. The following section focuses on the current interest and significance of this research area.

Significance and Current Interest

The practice of steganography for malicious purposes is a significant problem that has been prevalent in recent research and continues to evolve. Brunot (2019) finds that current types of malware delivered through steganography, especially malvertising and ransomware, are costing organizations billions of dollars. Malvertising, or malicious advertising, is the mixing of malware-based advertisements with legitimate online advertising; a user's system can be infected with malware by just having visited a page that contains malvertising (Malvertising, n.d.). Ransomware encrypts a victim's data files and demands payment before enabling decryption. Brunot also reports that the sophistication of attacks is constantly increasing, which puts pressure on the demand for equally sophisticated detection tools. In addition, other types of steganography threats continue to be a concern for organizations, including insiders hiding sensitive company information, illicit material stored on company resources, and criminal communications taking place on corporate websites (Brunot, 2019).

The number of stakeholders and practical use cases that are related to this research is potentially very large, since the illicit transfer of data affects virtually all private and public organizations. In fact, Zielinska notes that in our current technological state, the type of

carrier file for steganography could be not only an image file, but “any other file type or organizational unit of data… that naturally occurs in computer networks” (Zielinska et al., 2014). As a direct application, the proposed tool could be used by computer security technicians for regular data flow checkups, or to investigate specific incidents or suspicious files. For example, if employees start reporting unexpected, similar emails that contain random JPEG attachments, the security department might use the tool to attempt to identify a common origin or telling characteristics of these files, or to monitor internal communications that involve suspicious images, allowing early analysis before sensitive information can be leaked. One incident illustrating this internal use of images was cited by Brunot (2019): in 2018 malware infected the Magento e-commerce system and hid payment details inside product images published on its website. Finally, stakeholders such as general security researchers would also potentially benefit from the proposed tool, as they may use the artifact to test known steganography samples and possibly improve or replace the tool with something better. As Zielinska points out, the lack of a “‘one size fits all’ solution” should motivate researchers to develop more practical and immediately-usable steganalysis methods (Zielinska et al., 2014).

The consequences and ramifications of the malicious use of digital steganography can be serious and affect all stakeholders connected to an organization, and a clear need exists for further research and refinement of solutions to this problem. As part of the response to these issues, the current study is guided by the three research questions detailed in the next section.

Research Questions

The goals of design science research can be framed in terms of a design problem and related knowledge questions (Weiringa, 2014). For this study, the design problem can be stated as follows: Improve the area of JPEG steganalysis by designing a program that detects hidden data in JPEGs embedded by known tools, in order to help security professionals thwart malicious data-hiding activities. The following are knowledge questions related to this problem that this research endeavored to answer:

1. Within the scope of the embedding tools targeted by this study, what level of accuracy can be achieved by the program in terms of successfully detecting the presence of steganography in a given JPEG?
2. If steganography is detected in a JPEG, what level of accuracy can be achieved by the program in terms of successfully identifying which target tool was used to hide the data?
3. What kinds of obstacles and difficulties were encountered in terms of designing successful detection methods for the steganographic tools and embedding methods targeted by this study, and which (if any) of these obstacles were not overcome?

The documentation collected during the design process and validation experiment for the artifact of this study was used to answer these questions, and those answers are discussed in detail in Chapter 4. The next section describes the scope and limitations of this study.

Scope and Limitations

The scope of this study includes the design of a software tool (JRevealPEG) that performs detection of steganography on JPEG images processed by a preselected group of open-source embedding tools. The specific group of embedding tools that were included are discussed in detail in Chapter 3. Through an experiment, the detection accuracy of JRevealPEG was measured in relation to a selection of preprocessed JPEG images, and the processing times that elapsed during the experiment were also recorded. Difficulties and obstacles during the design process were also observed and documented.

There were several limitations to this study that should be noted. First, as a steganalysis tool, the functionality of JRevealPEG in this study is limited to JPEG images only. Other image types, such as Portable Network Graphics (PNG), Bitmap (BMP), and Tagged Image File Format (TIFF) are not considered valid files for analysis. Audio, video, and other file types are also excluded. Additionally, JRevealPEG is not intended as a universal steganalysis tool. Only the selected group of steganography tools listed in Chapter 3 are meant to be within the scope of its detection capabilities. Finally, while this research did involve implementation of steganography detection capability, the actual retrieval of hidden messages was not considered within scope.

Introduction: Summary

This introduction began by providing stories about the use of steganography throughout history and noting the variety of techniques employed. The main problem of this study was identified, namely the ongoing use of steganography for malicious purposes and the need for further research and solutions. The purpose of this study was described as the design and analysis of a new steganalysis tool written in Python, called JRevealPEG. The lack of effective existing tools and methods and the popularity of the JPEG image type were given as the main motivating factors for this study. It was noted that malware steganography is still a current and significant issue which costs organizations billions of dollars, and therefore interest in this area of research and the need for solutions continues to grow. The research questions stated for this study are concerned with the resultant accuracy of JRevealPEG in detecting steganography, its accuracy in identifying the responsible tool, and also documenting obstacles encountered during the research. Finally, only detection capability, JPEG images, and a selected group of open-source embedding tools are included in this study's scope. Hidden message retrieval, other tools, and other image and file types are outside the scope of this research.

Chapter 2 is a literature review surveying current research and methods in the realm of image steganography and steganalysis. It also provides a brief background on the JPEG image compression standard.

CHAPTER 2

LITERATURE REVIEW

Recent Efforts in General Image Steganography/Steganalysis

The art of hiding data in digital images in general, as well as its counterpart, the art of detecting, extracting, and/or reconstructing such data, have become highly technical areas recently, continually evolving with increasingly sophisticated techniques. This is evidenced by certain relevant studies of the past few years. The next several examples do not focus on particular image formats, but rather image steganography and steganalysis in general. The JPEG image type is of particular interest to this study and the steganography world overall, and many studies relating specifically to JPEG steganalysis will be addressed later.

Steganography Focus

Recent research into image steganography/steganalysis tends to focus on either the data hiding side (steganography), or the data detection and extraction side (steganalysis). On the steganography side, in a study by Das and Dhara (2018), the well-known least-significant-bit (LSB) substitution method of hiding data in an image's pixels is explored, but utilized a newly-proposed manner on gray-scale images, in combination with other techniques. In LSB, the least significant bits in the cover (original) image are replaced by the bits of the message that is to be hidden. The authors in this case first apply a custom extended local binary pattern (ELBP), which is a way of encoding image pixel data using blocks of 3x3 pixel values in the gray-scale image. First, ELBP converts each of the surrounding decimal pixel values in the 3x3 block (all but the center pixel) into a 3-bit binary code. After this, the secret message is embedded by the LSB technique into those 3-bit codes and the pixel block is converted back to decimal values. Finally, an algorithm called optimal pixel adjustment process (OPAP) is applied to reduce the distortion caused by LSB and improve the final image quality. The authors concluded that their technique allowed for a higher embedding rate than comparative methods, resulted in better image quality, and presented a high resistance against statistical steganalysis attacks (Das & Dhara, 2018).

Another study on the steganography side (Sairam & Boopathybagan, 2019) also attempts to improve hiding capacity and maintain good image quality, however that proposed method also involves a layer of encryption of the data. The study examines a technique that uses modulus values to find random locations for hiding bits of data in the cover image. The LSB technique mentioned above is also employed here. In this case, the proposed modulus method was found to be most effective when applied to the non-compressed bitmap image type (BMP). Like the study above, the authors also used RS steganalysis to test the security of their method (Sairam & Boopathybagan, 2019).

A steganography technique using a curvelet transform method is proposed in a paper by Subhedar and Mankar (2018), which tested grayscale images only. The curvelet transform algorithm that was used looks at an image geometrically in an effort to hide data more sparsely and effectively, ostensibly making it more resistant to detection. Additionally, the study discusses the importance of choosing appropriate cover images for more secure steganography and contributes a new technique for this purpose. The authors employ a technique they call "fuzzy logic" to sort through and analyze a collection of images to choose the best candidates. To evaluate the robustness of their proposed methods, the authors employ machine learning, in this case a support vector machine (SVM) classifier (Subhedar & Mankar, 2018).

Steganalysis Focus

While the topic of finding new ways of hiding data in images is a current and significant research area, equal if not greater research activity appears to be occurring in the realm of detecting that hidden data through image steganalysis. Two recent studies focus on data extraction targeting a well-known steganography technique called HUGO (Highly Undetectable steGO). In one study by Gan, J. Liu, Luo, Yang, and Liu (2018), the authors focus on an extraction technique to retrieve hidden data embedded by HUGO, but which has also been encrypted. HUGO is an adaptive steganography technique that can select the best pixels to use in order to create minimal distortion in the resulting image. The method proposed by Gan et al. claims to extract an encrypted hidden message only when some of the plain text is known, in this case the file format and length of the message. A key part of this method is stated as making use of syndrome-trellis codes (STCs), a concept from information

coding theory that can be used to determine places to embed messages in a cover image (Gan et al, 2018).

In the other paper targeting HUGO (Luo et al., 2016), the authors propose a blind analysis method to extract the hidden data, also making use of STC data. The difference here is that the hidden data is not encrypted, eliminating the need for known plain text. In this study, the authors construct all possible STC parameters, which they compare to the identified bits of hidden data in an effort to reconstruct the original message. Another notable part of this process is that it also employs machine learning, here in the form of an estimator that helps determine the relative payload (size of hidden message vs. size of cover image) of the hidden data (Luo et al., 2016).

A study by Malik, Subbalakshmi, and Chandramouli (2016) explores a statistical approach to detecting steganography hidden by a particular technique. Their proposed method examines a particular transformation of pixel data called quantization index modulation (QIM), and implements a kind of randomness test to detect steganography. The technique uses hypothesis testing in the form of decision rule formulas for the final determination of which part of an image is cover, and which part is steganography (Malik et al., 2016).

Machine learning appears yet again in a paper by Lu et al. (2019), which explores a new method of hidden message detection in binary images. Binary images are black and white images with two possible values (1 or 0) for each pixel. The proposed method first compiles histograms of certain pixel structural element (SE) patterns based upon predetermined criteria. From these SE patterns, a feature vector (set of relevant characteristics) is chosen to use with an SVM classifier for final determination (Lu et al., 2019).

JPEG Compression

Since the focus of the present study is the steganalysis of JPEG images, a brief background on the standard JPEG compression algorithm will be helpful. All of the steganalysis techniques discussed in later sections are focused on JPEG images, so a basic familiarity with the standard should aid greatly in understanding the relevant concepts and terminology.

JPEG (Joint Photographic Experts Group) technically does not refer to an image file format, but rather a compression method invented to be able to support continuous-tone

images. Continuous-tone images that have thousands (even millions) of colors resulting from the real-world origins of the subjects (JPEG Compression, n.d.). Because of the need for such a large number of colors, the image also has to be capable of large pixel depths, e.g. 24 bits would equate to 2^24, or over 16 million colors. In addition, it is mainly a lossy compression method, meaning it discards unnecessary data during encoding to reduce file size. In terms of an image, this means that the algorithm discards certain image data that cannot be seen by the human eye in any case (JPEG Compression, n.d.).

In the core standard, called Baseline JPEG, encoding is based on a category of mathematical operations called the Discrete Cosine Transform (DCT). In the third stage of compression, the DCT is applied to image data that has been divided into 8x8 pixel blocks (here, pixels are single values representing certain colors, rather than separate RGB values). The DCT values are then "quantized," or divided by values from a particular quantization table, called "quantization coefficients" (JPEG Compression, n.d.). The quantization step is where the "unnecessary" pixel data is discarded during compression. After that, however, the final encoding step is lossless, since it only involves removing redundant information (JPEG Compression, n.d.).

For the JPEG steganalysis examples discussed below, the compression stages above are the most relevant component of the JPEG algorithm. Overall, however, there are five stages in Baseline JPEG compression. A simplified diagram is provided in Figure 1 to illustrate this (JPEG Compression, n.d.). These steps are followed in reverse order to decode and display a JPEG-compressed image.

Raw Image ->	Optimize Colors ->	Downsample ->	DCT ->	Quantize ->	Encode ->	**JPEG Compressed**

Figure 1: Baseline JPEG Compression Stages

Regarding the byte structure of a compressed JPEG image file, the most pertinent aspects are addressed during discussions of the design of the artifact which is the object of this study. However, it may be useful to note a few basic elements here. JPEG images contain special two-byte markers, which mark the beginning of each particular segment of the file. Each of these markers begins with the value 0xFF. Also, there may be several segments before the segment containing the actual image bytes, such as those providing DCT-specific

information and the quantization tables used in the encoding (JPEG, n.d.). Analysis of the byte organization of a JPEG image file can be relevant to many types of steganalysis techniques.

JPEG Image Steganalysis: Recent Techniques

Machine Learning

The practice of employing machine learning algorithms and techniques to facilitate JPEG image steganalysis appears to be the dominant trend in the literature of recent years, probably due to its evident effectiveness. Often, a study will propose one or more new techniques to aid in determining the appropriate features to extract from a JPEG image set, then this data is fed to one or more machine learning classifiers for training and testing, and finally accuracy analysis. The DCT domain that is part of JPEG compression is a major theme and an important source of data for the majority of these steganalysis studies. Other common themes include targeting content-adaptive JPEG steganography, as well as drawing upon digital image filtering techniques when creating a steganalysis method.

DCT and Machine Learning

A method proposed by Jia-Fa, Xin-Xin, Gang, Wei-Guo, and Na-Na (2016) targets steganography that uses additive operations on AC coefficients to hide data. In a quantized 8x8 pixel block of a JPEG, there are two types of DCT coefficients that occupy those 64 cells: one cell is labeled as DC, and the other 63 are labeled as AC. The study by Jia-Fa et al. exploits the statistical changes in the AC coefficients that show up after steganography has occurred. Their experiments made use of JPEG images for both the cover and the stego (hidden) data. The feature vector they employed contained only three data points, and they used a Fisher linear classifier. It was concluded that their method was simpler and resulted in a lower false positive rate compared to other existing methods (Jia-Fa et al., 2016).

Nouri and Mansouri (2017) explored a technique that models natural image statistics using a method called singular value decomposition (SVD). SVD is a type of matrix decomposition method used in signal processing. The singular values (SVs) are then used to determine features for classification using an SVM binary classifier. For JPEGs analyzed in

this study, these features were extracted from the quantized DCT coefficients (Nouri & Mansouri, 2017).

Some JPEG images can be doubly compressed, which can cause issues for steganalyzers. This is addressed in a study by Yang, Kong, and Feng (2018), which attempts to improve detection performance by reducing the discrepancies between training and testing sets that occur due to double compression. Also making use of and image's DCT coefficients, the authors use a multi-classifier to detect the double compression initially. Through this they determine what are referred to as quality factors, which help adjust the features used in the training set that is fed to an SVM. The authors claim that in general, their technique is an improvement over comparative methods (Yang et al., 2018).

Another technique that analyzes DCT coefficients in a different manner was proposed by Rabee, Mohamed, and Mahdy (2018). In what is described as a blind steganalysis technique, the authors measure the differences between DCT coefficients that occur before and after cropping the image. Essentially the procedure is this: the initial DCT coefficients are extracted, the image is decompressed, it is cropped by four columns and four rows, it is recompressed, the new coefficients are extracted, and finally they are compared. This method uses an SVM classifier, and is tested against five known steganography algorithms. The authors conclude that their method generally performed better than a comparative method called Merged Features (Rabee et al., 2018).

Butora and Fridrich (2020) put forth a method called Reverse JPEG Compatibility Attack, which targets rounding errors in integer values used during the DCT stage of JPEG compression. The proposed method was compatible with both color and grayscale JPEGs, but limited to quality factors 99 and 100, which are the two highest compression qualities available with the JPEG algorithm. The authors used statistical hypothesis testing to initially evaluate their method, but stated that the best detection would result from the use of classifiers. Hence, they also used three classifiers and tested against five known steganography techniques. A notable observation was that the classifiers behaved somewhat "universally," in that they seemed to generalize detection to steganography methods they had not seen (Butora and Fridrich, 2020).

Content-adaptive Stego, Image Filters, and Machine Learning

One study by Denemark, Boroumand, and Fridrich (2016) focused on detection of content-adaptive steganography in JPEGs. Content-adaptive steganography describes a means of choosing the best locations for embedding hidden data in a cover image, as opposed to more random methods. Denemark et al. proposed a way to incorporate selection-channel-aware features into data for classifier training and steganography detection. The idea of a selection channel refers to the probability of certain cover image locations being changed during hidden data embedding. For small payloads in particular, the authors concluded that their method resulted in significant detection improvement (Denemark et al., 2016).

Content-adaptive steganalysis and digital image filtering is combined in a study by Song et al. (2017). The authors develop a characteristic called a Gabor Rich Feature (GRF), which they based on two-dimensional Gabor image filtering, where the JPEG is filtered after being decompressed. The final features are selected based on statistics, including histograms, and merging of other features. Their technique is tested on three current steganography methods using an ensemble classifier (multiple decision engines). The proposed GRF was concluded to improve detection, when compared to other types of features being used in the field (Song et al., 2017).

Feng, Zhang, Ren, Qian, and Li (2020) devised a special combination of digital image filtering techniques to compute JPEG image residuals, referring to traces of embedded hidden data that can be used in steganalysis. The filters used in this case were base filters and cascade filters, which have special properties in the signal processing domain. The computed residuals were used to generate features to be fed to an ensemble classifier for analysis. This method was tested against four known steganography methods, including a well-known one called J-UNIWARD. Various results were reported, based on particular configurations of the filters used (Feng et al., 2020).

One final paper that made explicit use of machine learning (in this case, a neural network) seemed to claim that using DCT data in the initial set up may actually hurt efficiency, in terms of a neural network analyzer (Boroumand, Mo, & Fridrich, 2019). The proposed method instead promoted deep learning from end to end, which means useful features were to be learned by the analyzer instead of being fed to it ahead of time. The authors developed what they called SRNet, or Steganalysis Residual Network. This technique

used grayscale images and targeted the UED-JC and J-UNIWARD embedding techniques. The authors also claimed that SRNet was the first neural network steganalyzer that did not require extra that information be given to it initially, considering other work referred to in the paper (Boroumand et al., 2019).

Non-machine-learning Techniques

Very few examples in recent literature appeared to propose JPEG steganalysis techniques that did not make use of machine learning as a key component. This lack of current research in JPEG steganalysis that does not involve large samples for machine learning is part of the motivation for the software artifact being proposed in the present study. The following two studies describe recent attempts at this type of JPEG steganalysis.

Rather than detection, a study by Xu, Liu, Gan, and Luo (2018) explored a new method to aid in the extraction of hidden messages in JPEGs. The authors propose a technique to recover the stego key, which is a seed value for generating a pseudorandom number in steganography. The seed leads to a random path being picked for embedding the bits of hidden data in the DCT coefficients of a JPEG, using the LSB technique. The paper's detection method uses hypothesis testing to compare the statistics from different samples of DCT coefficients, trying to recover the original embedding path and derive the stego key. The stego key could then theoretically be used to extract the hidden data. The proposed method was tested against F5 and OutGuess steganography methods and found to perform more quickly and with less computational complexity, as compared to one other competing detection method (Xu et al., 2018).

Qiao et al. (2019) proposed an adaptive steganalysis framework for JPEG steganography, based on a statistical model of quantized DCT coefficients. This framework also relies on hypothesis testing to detect steganographic data, as opposed to a machine-learning classifier. For the technique discussed in this study, the authors assume that the stego data was embedded using LSB replacement in the quantized coefficients. In order for the proposed framework to have the best performance, high accuracy is necessary concerning three main factors: the statistical model used, the distribution parameters, and the payload size estimation. The authors' evaluation consisted of comparing the performance of their framework to machine-learning classifiers in two types of scenarios. Tested against non-

adaptive embedding methods, the framework performed better than machine-learning classifiers. However, tested against modern adaptive methods, such as J-UNIWARD, both the framework and the classifiers were ineffective, especially with small payloads (Qiao et al., 2019).

Literature Review: Summary

This review has outlined the various efforts in recent research of image steganography and steganalysis in general, a brief introduction highlighting the essential components of JPEG compression, and a survey of recent steganalysis research focused specifically on JPEG images. It was observed that the majority of current JPEG steganalysis techniques involve some type of machine learning, as well as various JPEG-specific concepts, such as the DCT domain, content-adaptive steganography, and signal processing techniques such as image filtering. It was also noted that there is far less current research on JPEG steganalysis without the use of machine learning, and that the methods that are proposed are limited as to the steganographic methods they target, as well as the initial assumed conditions. It is clear that much room exists for further research in the area of JPEG steganalysis without the aid of machine learning, which is the focus of the present study.

Table 1 lists and categorizes the cited references as they appear in this review:

Category	**Citation**
Recent Efforts in General Image Steganography/Steganalysis	
• *Steganography Focus*	Das & Dhara, 2018
	Sairam & Boopathybagan, 2019
	Subhedar & Mankar, 2018
• *Steganalysis Focus*	Gan et al, 2018
	Luo et al., 2016
	Malik et al., 2016
	Lu et al., 2019
JPEG Compression	JPEG Compression, n.d.
	JPEG, n.d.
JPEG Image Steganalysis: Recent Techniques	
• *Machine Learning (ML)*	
• *DCT and ML*	Jia-Fa et al., 2016

	Nouri & Mansouri, 2017
	Yang et al., 2018
	Rabee et al., 2018
	Butora and Fridrich, 2020
• *Content-adaptive Stego, Image Filters, and ML*	Denemark et al., 2016
	Song et al., 2017
	Feng et al., 2020
	Boroumand et al., 2019
• *Non-machine-learning Techniques*	Xu et al., 2018
	Qiao et al., 2019

Table 1: List of Citations in Literature Review

The next chapter describes the methodology used to develop the JRevealPEG artifact, which was design science. The Python-language architecture of the artifact and its features are discussed in great detail, and the detection functionality of the program in relation to each of the target steganography tools is explained thoroughly.

CHAPTER 3

THEORY AND ARTIFACT DESIGN

The main objective of this research was to design a software steganalysis artifact that is able to detect hidden data in JPEG images, targeting a specific group of current, open-source tools. The artifact should be able to detect the presence of steganography embedded by the target tools, while at the same time identifying the tool that was used. The overall methodology employed for this study is design science, referencing the framework explained by Weiringa (2014).

As stated in the introduction, Weiringa (2014) suggests that the main goal of design science research can be thought of as a design problem and its related knowledge questions. The design problem for this study was stated as follows: Improve the area of JPEG steganalysis by designing a program that detects hidden data in JPEGs embedded by known tools, in order to help security professionals thwart malicious data-hiding activities.

The related knowledge questions are also restated below:

1. Within the scope of the embedding tools targeted by this study, what level of accuracy can be achieved by the program in terms of successfully detecting the presence of steganography in a given JPEG?
2. If steganography is detected in a JPEG, what level of accuracy can be achieved by the program in terms of successfully identifying which target tool was used to hide the data?
3. What kinds of obstacles and difficulties were encountered in terms of designing successful detection methods for the steganographic tools and embedding methods targeted by this study, and which (if any) of these obstacles were not overcome?

Within design science methodology, the design cycle can be divided into three categories: problem investigation, treatment design, and treatment validation (Weiringa, 2014). The first two categories are discussed in this chapter in relation to the present study.

The third category, treatment validation, is addressed in Chapter 4 along with the discussion of the results from a single-case mechanism experiment.

Problem Investigation

According to Weiringa (2014), an important part of problem investigation involves identifying a conceptual framework and key concepts related to the problem. Most of the background information that would help establish a conceptual framework for the research problem is discussed in the Introduction and Literature Review. However, the main relevant concepts can be summarized here.

One of the main higher-level concepts relevant to this project is steganalysis, along with its sibling term, steganography. Steganography in general is the act of hiding messages or other data inside of other messages or data, and steganalysis is the detection and/or recovery of such hidden data. In terms of digital media, stenography can be performed with text, image, audio, and video. This study focuses on image steganalysis, specifically JPEG steganalysis. Hence, a good understanding of the JPEG image specification and related terms (discussed in the Literature Review) is also essential to the research problem.

There have been many steganographic algorithms developed for the JPEG format, some of which are highly technical and difficult to understand. In response, a variety of detection techniques have also been proposed that focus either on certain algorithms, or on universal detection. Instead of delving deeply into several highly technical algorithms or taking on the burden of developing another universal detection system, the program developed in this research focuses on a small group of current, open-source software tools that are freely available for anyone to use. The tools that were chosen for this study are listed in the next section, and later each is examined and discussed in terms of its steganographic technique and related program architecture. As will become apparent, an intimate understanding of these target programs was essential to the successful design of the main artifact of this study.

Treatment Design

In the context of design science, the term “treatment” can be used to describe “the interaction between the artifact and the problem context,” which, in this case, refers to the

proposed program interacting with JPEG input to detect hidden data (Weiringa, 2014, p. 28). The resulting architecture of the treatment proposed in this study is meant to address the determined requirements, and these are based on the design science research goals, as stated above. For the development environment, the operating system used was Windows 10, and the programming language was Python 3. The following is a list of the initial feature requirements for the program, named JRevealPEG:

1. Must run through a command-line interface.
2. Must provide a menu for the user that lists all commands.
3. Must be able to take JPEG images of varying sizes and dimensions as input.
4. Must calculate processing times for each input file.
5. Must provide a detection report (detection positives and negatives with tools identified, processing times, other relevant data) as output to the screen.
6. Must save reports to log files (text and csv).
7. Must implement user input validation and exception handling.

As is common for a program written in Python, the architecture involves multiple Python script files. The program makes use of several custom functions and modules, as well as a handful of standard Python libraries. The design and functionality of each component are discussed in detail in the following section, including how JPEGs are processed and how steganography is detected in relation to each of the target software tools.

The software tools targeted by the artifact in this study are *Jsteg 0.3.0* as released on 8/16/2018*, SteganPEG 1.0* as released on 1/5/2011*,* and *OutGuess 0.2.2* as released on 1/20/2019 (Abhiram, 2011; Champine, 2018; Filho, 2019). This list was developed by first choosing a group of several candidate programs through a moderately vigorous Internet search, intended to simulate a selection of tools that would be freely available and appear for any user to implement. From the initial list, some candidates were eliminated on account of discovered incompatibilities with the required JPEG format. Others were removed because of age, compilation problems, other bugs, or limited message-embedding capacity. Through further analysis, the resulting list of target programs was found to include a promising variety of complexities, challenges, and steganographic methods that would fit the scope and purpose

of this research appropriately. The specific methods employed by these programs are addressed during the discussion of the JRevealPEG architecture, below.

It is also worthwhile to clarify the scope of JRevealPEG by identifying some notable functional limitations. First, many existing methods employ machine learning and classifiers to detect steganography. This program does not use machine learning. Next, the tool does not endeavor to have the ability to detect hidden data spread across multiple images. Third, visual image inspection (by a human) is not a factor in hidden data detection, i.e. detection occurs solely within the software. Finally, the program will not attempt to extract or reassemble the hidden data, unless this becomes a byproduct of a particular detection algorithm.

JRevealPEG Architecture and Components

In this section, the components and architecture of the artifact of this study, JRevealPEG, are discussed. At the same time, the steganographic methods and structures of the three targeted embedding tools are examined in relation to the corresponding detection techniques developed in the artifact.

Overview of JRevealPEG Structure and Behavior

JRevealPEG is composed of one main module, jrpeg.py, and four other custom modules used as imports in the main module: coefx.py, jsdec.py, spdec.py, and ogdec.py. The main module contains the entrance point for the program, which can be executed in a command window and takes no arguments. Upon execution, a simple menu appears to the user which allows the choice of entering the path to a single file, entering the path to a folder, bringing up the help page, or quitting the program (see Figure 2).

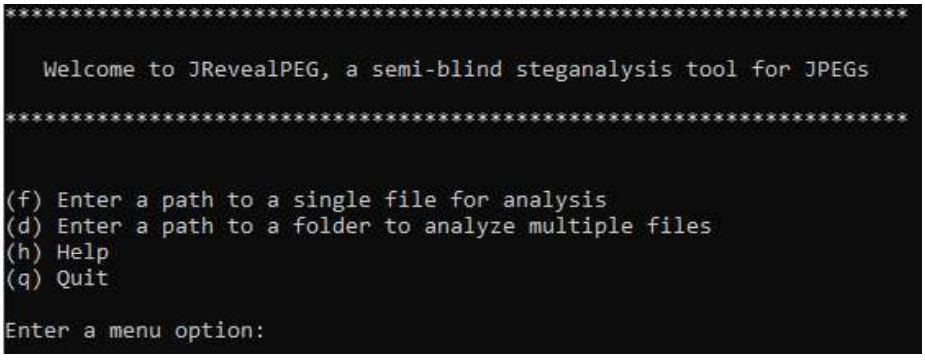

Figure 2: JRevealPEG Main Menu

If either analysis option is selected, the main detection sequence will begin for the chosen input file(s). As each file is analyzed, a report is displayed to the console, which includes information on the file currently being analyzed, the results of each stage of detection per target program, and processing times for initial JPEG analysis and the total detection sequence for that file. The detection sequence begins with the first target program (Jsteg), and if detection is negative it moves on to the next target (SteganPEG), and if negative again, the final target will be checked (OutGuess). The assumption is that once an input file tests positive for steganography by one of the target tools, there is no need to check the other targets, so the detection sequence will skip to completion for that file as soon as a positive result is attained. If no positive result is found, there will be a message confirming that status (see Figures 3 and 4).

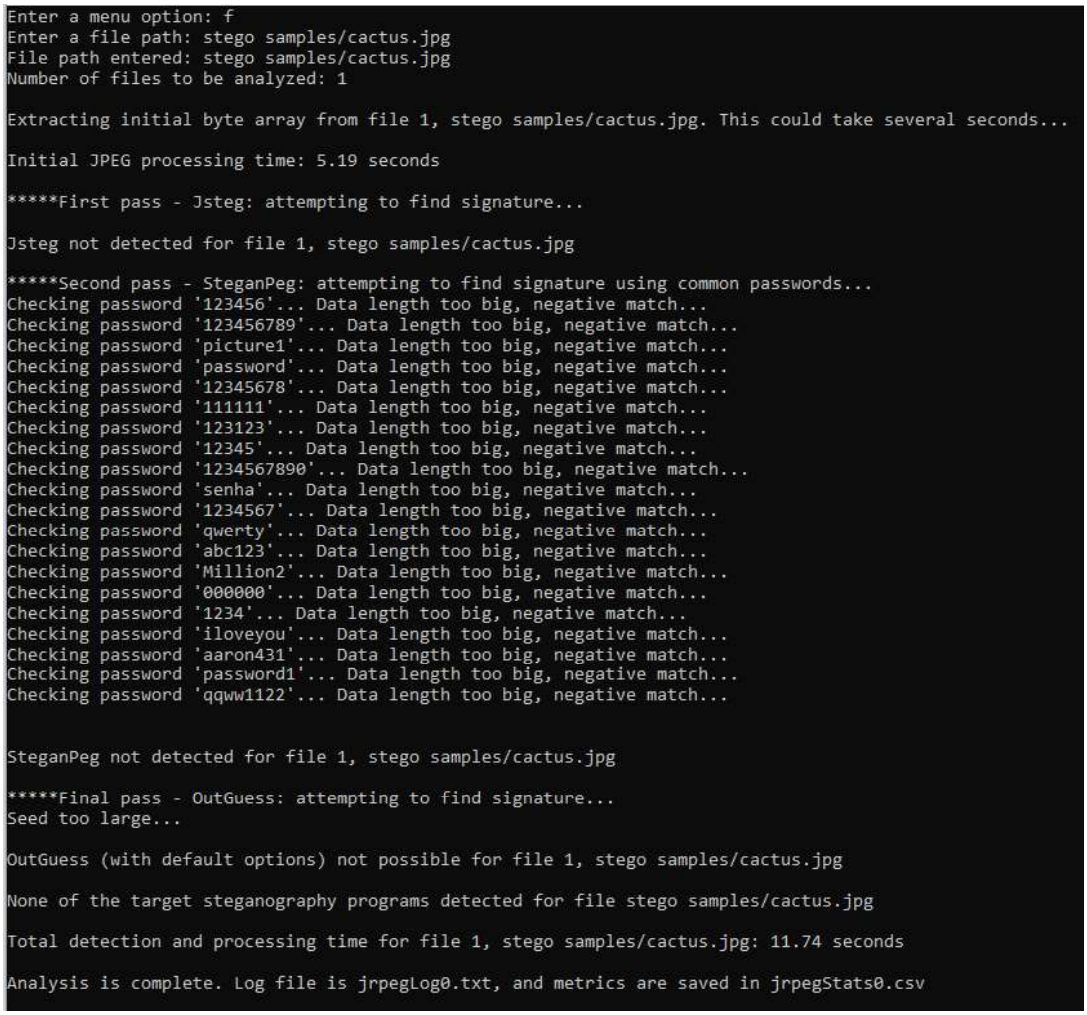

Figure 3: Sample Output – No Positive Result

```
Enter a menu option: f
Enter a file path: stego samples/jsCowMsd.jpg
File path entered: stego samples/jsCowMsd.jpg
Number of files to be analyzed: 1

Extracting initial byte array from file 1, stego samples/jsCowMsd.jpg. This could take several seconds...

Initial JPEG processing time: 7.53 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 1, stego samples/jsCowMsd.jpg

Total detection and processing time for file 1, stego samples/jsCowMsd.jpg: 9.23 seconds

Analysis is complete. Log file is jrpegLog1.txt, and metrics are saved in jrpegStats1.csv
```

Figure 4: Sample Output – One Positive Result

JRevealPEG also has a logging function which automatically saves the detection sequence report to a text file. In addition, a CSV file is created that contains a list of all JPEG files analyzed, their sizes and dimensions, positive and negative test results, and both types of processing time (see Figure 5).

	A	B	C	D	E	F	G	H	I
1	File	Size	Height	Width	Jsteg	SteganPEG	OutGuess	CoeffTime	OverallTime
2	jsGoldHost.jpg	485007	3300	3300	TRUE	FALSE	FALSE	4.58	6.22
3	ogGoldHost.jpg	488385	3300	3300	FALSE	FALSE	TRUE	4.84	13.18
4	spGoldHost.jpg	796987	3300	3300	FALSE	TRUE	FALSE	6.98	10.19

Figure 5: Sample CSV file

The other four modules are imported into jrpeg.py and are responsible for the JPEG processing and detection algorithms that occur during the main detection sequence. The coefx.py module performs a partial decompression of the JPEG file and returns the relevant bytes for steganography detection. The modules jsdec.py, spdec.py, and ogdec.py each perform a customized detection algorithm targeting the steganography programs Jsteg, SteganPEG, and OutGuess, respectively. All five modules are discussed in more detail below, along with relevant analyses of the three target steganography tools.

The Main Module: jrpeg.py

Since all of the costly processing is done by the custom external modules, the main module of JRevealPEG is fairly lightweight. The two most important features of jrpeg.py are its ability to coordinate the requisite files to be analyzed, and to populate and save the log

files. Along with the four custom modules mentioned above, jrpeg.py imports the standard Python time and os modules for timestamping and filesystem manipulation purposes.

In order to facilitate a specific kind of logging capability, a custom class called Logger was created to save the two types of analysis reports mentioned above, in this case a text file and a CSV file. Logger contains a method called lprint(), which is used to simultaneously print to the screen and save to the log text file (see Figure 6).

```
163     def lprint(self, line):
164         print(line)
165         self.logFile.write(line + "\n")
```

Figure 6: Method Logger.lprint() from jrpeg.py

The detection sequence is handled and directed by the function analyze() (see Figure 7). This function is set up to handle a list of one or more input files, depending on what has been selected by the user beforehand. Exception handling is incorporated throughout the detection sequence, the steps of which are as follows:

1. Save the starting time.
2. If the list is not empty, process the next input file using coefx.py.
    a. Else, go to Step 11 to end analysis.
3. If no errors, save the JPEG processing time for current file.
    a. Else, go to Step 1 for next input file.
4. Initialize all detection results to false.
5. Using jsdec.py, apply Jsteg detection algorithm to processed JPEG data.
6. If Jsteg result is false, apply SteganPEG detection algorithm using spdec.py.
    a. Else, update result message, go to Step 8 to complete sequence for file.
7. If SteganPEG result is false, apply OutGuess detection algorithm using ogdec.py.
    a. Else, update result message, go to Step 8 to complete sequence for file.
8. Save the total processing time for current file.
9. Update user on final result and write stats to CSV file.
10. Go to Step 1 for next input file.
11. Display completion message and names of log and CSV files.
12. Close log text and CSV files.

```
def analyze(srcFiles, log):

    log.lprint("Number of files to be analyzed: " + str(len(srcFiles)))

    count = 0
    for f in srcFiles:
        count += 1

        #Extract array of Huffman-decoded scan data from input image (plus height, width -> y, x)
        log.lprint("\nExtracting initial byte array from file " + str(count) + ", " + f + ". This could take several seconds...")

        startTime = time.perf_counter()

        try:
            coeffs, y, x = coefx.extract(f)
            if (coeffs == None):
                log.lprint("\tNo scan data! May be bad file type.")
                continue
        except FileNotFoundError:
            log.lprint("Error: File not found! : " + f)
            continue
        except Exception as inst:
            log.lprint("Error: File might not be an accepted type of JPEG (baseline, 4:2:0 (2x2) subsampling) : " + f)
            m,n = inst.args
            log.lprint("\t" + m + " " + n)
            continue

        jpgProcTime = time.perf_counter() - startTime
        log.lprint("\nInitial JPEG processing time: {:.2f} seconds".format(jpgProcTime))
```

Figure 7: Beginning of function analyze() from jrpeg.py

Custom JPEG Processing: coefx.py

JRevealPEG employs custom steganography detection algorithms which have been tailored specifically for the three target programs of this study: Jsteg, SteganPEG, and OutGuess. However, before those algorithms can be applied to a JPEG, a very particular set of data must be decoded and extracted from the image file. Each of the steganography programs used in this study employs the least-significant-bit (LSB) substitution technique referenced in the Literature Review, which means the data to be hidden has to be placed into the image data *after* the lossy compression stages (DCT and quantization), but *before* the lossless compression step, namely the Huffman coding process (briefly explained below). Therefore, each JPEG only needs to be partially decompressed to the pre-Huffman coding state, the bytes of which contain the modified bits representing the hidden data, if any.

Several Python libraries exist that perform various types of JPEG manipulation, however it was found that most of the operations provided by these modules were not granular enough to be useful in this case. Only fully-decompressed JPEG data was normally available using the provided extraction functions. One recently-developed Python script was found, though, that could potentially aid in the necessary partial decompression, but it would need to be adapted and modified. The Baseline JPEG Decoder by Khalid (2019) is an experimental JPEG decoder written in Python as a single script, and as such several of its

functions exhibited promising granularity and became the basis of the coefx.py module in JRevealPEG.

The main purpose of the coefx.py module is to retrieve a JPEG's pre-Huffman image scan data, also known as the quantized DCT coefficients, for steganalysis. There were three main modifications/additions that needed to be made when adapting the Baseline JPEG Decoder for this purpose. The nature of each modification is introduced initially, then they are discussed in conjunction with code examples.

First, it turns out that Khalid's decoder (2019) only decompresses and displays Baseline JPEGs with no chroma subsampling, which is a method of saving space when encoding images by reducing the resolution of the color components, since humans do not notice differences in color as well as they do differences in luminance, or brightness (Chroma subsampling, n.d.). By contrast, two out of the three target steganography tools (Jsteg and OutGuess) only output JPEGs with what is known as 4:2:0, or 2x2 subsampling, which is common. Therefore, one necessary modification made for coefx.py was to ensure compatibility with 2x2 subsampling.

Another discovery, related to JPEG file segment markers, prompted the need for a second modification. It was mentioned in the Literature Review that JPEG files have varying types of segments delineated by special two-byte markers, each beginning with 0xFF. During decompression, it is necessary to recognize several of these markers in order to extract and use essential decoding parameters and tables. One of the most pertinent of these markers defines the quantization tables (DQT) and is made up of the bytes 0xFFDB; another defines the Huffman tables (DHT) and is 0xFFC4. For nearly all the color JPEGs with three components (Y, Cb, and Cr, for one luminance and two chrominance components) seen in this research, two quantization tables and four Huffman tables exist, each table having its own copy of the appropriate marker. However, it was observed that the JPEGs processed by the Jsteg tool only have one marker for each group of tables. Since Khalid's decoder (2019) cannot process the type of marker organization used by Jsteg, this capability was included in the code of coefx.py in relation to the DQT and DHT markers. Using 010 Editor, one can easily see the difference between the two types of marker organization from each kind of JPEG (see Figure 8).

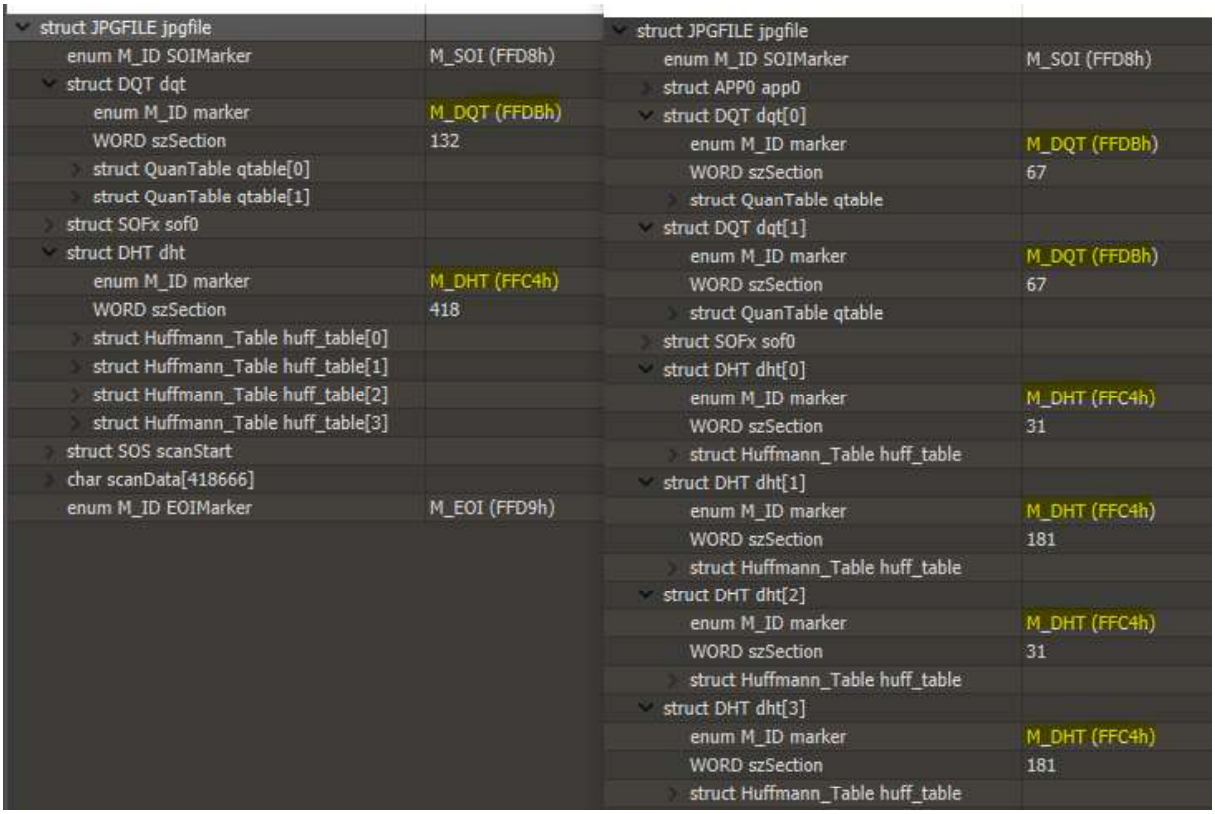

Figure 8: One marker per group (left) vs. one marker per table (right)

The third necessary modification has to do with extracting the partially-decompressed image data, as opposed to the fully-decompressed image pixels. Specifically, the data bytes right before the Huffman coding stage of JPEG compression were needed, since that is where LSB steganography occurs. Huffman coding is an algorithm that performs lossless compression of data by eliminating redundancies, thereby saving space without losing information. Huffman-encoded data can be most easily understood as a binary tree data structure, however implementing the decoding algorithm can be a bit more complicated. Fortunately, Khalid's decoder (2019) already implements Huffman decoding in Python, but even so, it is woven continuously into full JPEG decompression. It was necessary to add code that extracts the pre-Huffman bytes before they are allowed to be fully decompressed.

The module coefx.py retains enough of Khalid's code (2019) to facilitate the first stage of decompression, without needing to fully decompress and display the JPEG. The modifications mentioned above are incorporated into existing and new functions, and the bytes needed for steganography detection are exported. Regarding the first modification, that

which accommodates 2x2 chroma subsampling, the key was learning that the data bytes for the image components (Y, Cb, and Cr) are stored in a specific manner that differs from other subsampling specifications. Every luminance and two chrominance components (brightness and color) is represented as an 8x8 grid of pixel values; these three channels combine to form the full color image. Normally, each of these 8x8 combinations is a minimum coded unit (MCU). Each MCU is converted to a linear array of bytes. These arrays are stored sequentially from left to right and top to bottom as found in the displayed grid of a two-dimensional image. This results in an image component storage pattern of (YCbCr)(YCbCr)-etc. However, with 2x2 subsampling, the color components are sampled less often and averaged. Specifically, the MCU is 16x16 instead of 8x8, where the storage pattern is (YYYYCbCr)(YYYYCbCr)-etc. The existing method JPEG.StartOfScan() was heavily modified to account for this, as shown in Figure 9. A nested for loop iterates through an image grid, accounting for a 16x16 pixel MCU and the component pattern YYYYCbCr. The method BuildMatrix() is discussed below. Note that unconventional capitalizations were retained from preexisting code.

```
234     def StartOfScan(self, data, hdrlen):
235         data, lenchunk = RemoveFF00(data[hdrlen:])
236 
237         #Pad scan data with extra 0 byte in case SteganPeg is short
238         data.append(0)
239 
240         st = Stream(data)
241         oldlumdccoeff, oldCbdccoeff, oldCrdccoeff = 0, 0, 0
242 
243         #List of decoded MCU matrices
244         mcuList = []
245 
246         #Configured for 4:2:0 chroma subsampling only
247         for y in range(self.extHeight // 16):
248             for x in range(self.extWidth // 16):
249 
250                 #First decode 4 Y components in a row
251                 for i in range(4):
252                     cBlock, oldlumdccoeff = self.BuildMatrix(
253                         st, 0, self.quant[self.quantMapping[0]], oldlumdccoeff
254                     )
255                     mcuList.append(cBlock)
256 
257                 #Now decode Cb, Cr components
258                 cBlock, oldCbdccoeff = self.BuildMatrix(
259                     st, 1, self.quant[self.quantMapping[1]], oldCbdccoeff
260                 )
261                 mcuList.append(cBlock)
262 
263                 cBlock, oldCrdccoeff = self.BuildMatrix(
264                     st, 1, self.quant[self.quantMapping[2]], oldCrdccoeff
265                 )
266                 mcuList.append(cBlock)
267 
268         return mcuList
```

Figure 9: JPEG.StartOfScan

In order to fix the problem related to JPEG segment markers, coefx.py needed the ability to scan both styles of DQT and DHT marker organization, as mentioned above. Essentially, code was added to check the length of the data chunk after each marker, and this was used to determine the appropriate scanning algorithm for that segment. For the DQT case, a few lines of code were added to JPEG.decode() (see Figure 10). It turns out that the quantization tables were not essential to the functionality of the final artifact, but the code that handles this was retained as it may become useful in future work. As there are four Huffman tables to scan, the code that handles the DHT marker was more complex and seemed best as a new method, JPEG.checkHtStructType() (see Figure 11).

```
298                 if marker == 0xFFDB: #Quantization Table(s)
299                     chunk1 = chunk[:65]
300                     self.DefineQuantizationTables(chunk1)
301                     #If length is more than 69, only 1 marker for all tables, keep processing
302                     if (len_chunk > 69):
303                         chunk2 = chunk[65:130]
304                         self.DefineQuantizationTables(chunk2)
```

Figure 10: DQT marker scanning in JPEG.decode() from coefx.py

```
205     def checkHtStructType(self, len_chunk, data):
206         tableLengths = data[5:21]
207         valSum = 0
208         for val in tableLengths:
209             valSum += val
210         if ((len_chunk - 0x13) == valSum):
211             len_chunk += 2
212             chunk = data[4:len_chunk]
213             self.decodeHuffman(chunk)
214             return len_chunk, chunk, data
215         else:
216             #Only first DHT has the marker (0xFFC4)
217             tableLengths = data[5:21]
218             len_chunk = valSum + 0x13 + 2
219             chunk = data[4:len_chunk]
220             self.decodeHuffman(chunk)
221             data = data[len_chunk:]
222             for i in range(3):
223                 tableLengths = data[1:17]
224                 valSum = 0
225                 for val in tableLengths:
226                     valSum += val
227                 len_chunk = valSum + 0x13 - 2 #header adjustment
228                 chunk = data[:len_chunk] #start at index 0 cause no header here
229                 self.decodeHuffman(chunk)
230                 data = data[len_chunk:]
231             return len_chunk, chunk, data
```

Figure 11: JPEG.checkHtStructType from coefx.py

In order to achieve the final goal of pre-Huffman MCU extraction, it was necessary to identify that exact point of partial decompression in the original decoder, somewhere in the middle of the full decompression algorithm. Analysis showed that this occurred in the existing method JPEG.BuildMatrix(). Originally, JPEG.BuildMatrix() returned a fully-decompressed MCU component which went on to become part of the image display. For the purposes of the present artifact, the method was altered to stop decompression as soon as the pre-Huffman values are retrieved for a given component, and the MCU component block is returned as a one-dimensional list (see Figure 12).

```
136     def BuildMatrix(self, st, idx, quant, olddccoeff):
137         #Current block of MCU coefficients
138         cBlock = [0] * 64
139 
140         code = self.huffman_tables[0 + idx].GetCode(st)
141         bits = st.GetBitN(code)
142         dccoeff = DecodeNumber(code, bits) + olddccoeff
143 
144         cBlock[0] = dccoeff
145 
146         l = 1
147         while l < 64:
148             code = self.huffman_tables[16 + idx].GetCode(st)
149             if code == 0:
150                 break
151 
152             # The first part of the AC key_len
153             # is the number of leading zeros
154             if code > 15:
155                 l += code >> 4
156                 code = code & 0x0F
157 
158             bits = st.GetBitN(code)
159 
160             if l < 64:
161                 coeff = DecodeNumber(code, bits)
162 
163                 cBlock[l] = coeff
164 
165                 l += 1
166 
167         return cBlock, dccoeff
```

Figure 12: JPEG.BuildMatrix() from coefx.py

One last limitation of Khalid's original program (2019) should be mentioned, which was addressed in coefx.py. In the original decoder, only JPEGs with height and width

dimensions that are evenly divisible by eight could be decoded. The coefx.py module has the ability to decode JPEGs of any height and width dimensions. This problem was solved by simply rounding each dimension to the next higher multiple of sixteen, if not already divisible, which accounts for how irregularly-sized JPEGs are compressed. This dimension divisibility issue also has specific ramifications involving OutGuess in particular.

Finally, the main module of the artifact makes use of coefx.py by calling the entrypoint function extract() during the detection sequence. Note that along with a list of lists containing the MCU data, the height and width of the image is returned to the caller. These values become useful later in the detection sequence.

Jsteg and jsdec.py

Once the collection of pre-Huffman coefficients (the MCUs) is successfully extracted from a JPEG file using coefx.py, it can be passed to the first detection algorithm of the sequence, encapsulated by the module jsdec.py. The steganography tool targeted by jsdec.py is Jsteg, and this was chosen to be first in the detection sequence because the detection algorithm is the quickest and least complex of the three. First the relevant features and behavior of Jsteg itself are discussed, followed by an explanation of the jsdec.py detection module.

Jsteg (Champine, 2018) is written in the Go programming language, an object-oriented language invented by Google with C-like syntax (Go (programming language), n.d.). The program contains several modules and runs with a command-line interface. Jsteg uses the LSB substitution method for its steganography, as do the other two tools targeted in this study. Code tracing revealed which bytes are used to hide data, and which bytes are avoided. For all the target tools, it is essential to JRevealPEG's detection strategy to understand exactly which bytes are used for embedding.

In Jsteg's case, the program only hides data in the LSBs of Y-component bytes of an image's MCUs. There is further filtering inside each component as well. Recall that each MCU component can be thought of as an 8x8 grid of values. Through mathematical transformations, these values determine what is eventually displayed on a screen for each 8x8 section of an image's pixels. The upper left value in an 8x8 grid has the biggest impact on the image, and it is referred to as the DC coefficient. The other 63 values are called AC

coefficients (JPEG, n.d.). It is in the LSBs of the AC coefficients that Jsteg hides data, also noting that any byte values of -1, 0, and 1 are skipped (Champine, 2018).

Since any useful steganography program needs to be able to retrieve the data it hides, it has to have some method of identifying that data when it is time for extraction. Some tools require or at least have the option of using a password, which is specially encoded along with the hidden message; often the length of the message is embedded as well. The tool can then check for that password when asked to decode the message, failing if not matched. Jsteg does not require or have an option for a password, however it does use its own internal "magic" key, in this case the string "jsteg" (Champine, 2018). Analysis shows that the ASCII values of the characters of "jsteg" are prepended as the first five bytes of the hidden message, the bits of each byte stored in the order of least to most significant.

Given the knowledge of which bytes are used by Jsteg for steganography, as well as how the internal key is stored, the detection strategy to employ in the jsdec.py module became fairly straightforward:

1. Extract the first 40 eligible LSBs from the MCU list, Y components only.
2. Compare those bits to the "jsteg" key bits.
    a. If the bits match, return True.
    b. Else, return False.

Initially, jsdec.py uses the function filterMCUs() to retain only the Jsteg-eligible bytes for analysis, based on the tool's behavior as discussed above (see Figure 13). All Cb and Cr components and DC's are ignored, and all values of -1, 0, and 1 are filtered out.

```
31 #Filter out DC coeffs, -1, 0, 1; delete Cb, Cr components (every 5th and 6th sublist)
32 def filterMCUs(coeffs):
33     res = []
34     count = 1
35     for c in coeffs:
36         if ((count + 1) % 6 == 0 or count % 6 == 0):
37             count += 1
38             continue
39         c = c[1:] #Ignore DC coeff
40         c = list(filter((-1).__ne__, c))
41         c = list(filter((0).__ne__, c))
42         c = list(filter((0.0).__ne__, c))
43         c = list(filter((1).__ne__, c))
44         res.extend(c)
45         count += 1
46 
47     return res
```

Figure 13: Function filterMCUs from jsdec.py

Next, the function magic() collects the LSBs from the first 40 bytes in the filtered list and compares them to the “jsteg” ASCII bits, which are hardcoded in an array in the appropriate order (see Figure 14). If all the bits match, magic() returns a positive (true) detection result, and this result is passed back to the main module.

```
21 #8it order is reversed for 'jsteg' magic chars
22 jstegKey = [
23                 [0,1,0,1,0,1,1,0],
24                 [1,1,0,0,1,1,1,0],
25                 [0,0,1,0,1,1,1,0],
26                 [1,0,1,0,0,1,1,0],
27                 [1,1,1,0,0,1,1,0]
28             ]
```

```
49 #Compare extracted bits to magic bits
50 def magic(coeffs, key):
51     bitNum = 0
52     for i in range(5):
53         temp = []
54         for j in range(8):
55             temp.append(coeffs[bitNum]%2)
56             bitNum += 1
57         if (temp != key[i]):
58             return False
59
60     return True
```

Figure 14: Jsteg signature detection in function magic() from jsdec.py

SteganPEG and spdec.py

If jsdec.py returns a negative result and Jsteg is not detected, the main detection sequence passes the JPEG data to spdec.py, which contains the detection algorithm that targets SteganPEG. This was chosen as the second target in the sequence because it has a fairly concrete signature, but the detection strategy is more complex than for Jsteg. Also, it makes sense for the third position to go to OutGuess, as it has the vaguest signature and its detection algorithm should only be triggered if the first two come back negative.

SteganPEG (Abhiram, 2011) is written in Visual Basic and runs through a graphical user interface, rather than the command line. The program’s steganography procedure is quite a bit more complex than Jsteg, as it not only requires a password, but compresses and encrypts the data before hiding it. Like both Jsteg and OutGuess, SteganPEG does use LSB substitution to store the final data bits, however it does this in a slightly different manner than the other two tools. Additionally, and unlike Jsteg and OutGuess, SteganPEG preserves the

subsampling ratio of the original cover image. However, for the best overall compatibility and proof-of-concept purposes, it was decided that JRevealPEG will only process 2x2 subsampled JPEGs in its initial incarnation.

To prepare for encryption, SteganPEG applies a special encoding function to the password received from the user that transforms it into an array of integers. This password array is used to pseudo-randomly determine how each byte of data will be encrypted, via a bit rotation sequence. Also, two special values are concatenated with the compressed message data. First, a checksum is generated by a function involving the bytes of message data and appended to the data. Then, the length of the message data is prepended to the message data as four header bytes. The final result is then encrypted using the rotation sequence mentioned above (Abhiram, 2011).

As for which LSBs are modified during embedding, SteganPEG considers all MCU components fair game (both luminance and chrominance), but it does skip the DC coefficients and zero-value bytes. One last critical realization yielded by analysis was that the bit values of the message data get flipped when embedded in negative bytes. In other words, when decoding the message, negative odd and positive even bytes yield 0s, and the others yield 1s.

The detection strategy employed in the spdec.py module needed to consider all the mentioned complexities in SteganPEG's algorithm except the compression of the message data, since full message extraction is not a goal in this study. Also, since SteganPEG requires a password, it was decided this first version of the detector would assume the password is from a known list; a fully-blind, brute-force version might be a possibility for future work and is addressed in the Conclusion chapter.

Because there were several layers of complexity that needed to be reversed in order to identify a possible SteganPEG signature, much of the detection code in spdec.py was the result of isolating and adapting key routines from the original Visual Basic source code, and replicating and modifying it in Python. The overall detection strategy in spdec.py can be summarized as follows:

1. Extract the eligible bytes that could contain embedded data from the MCU list.
2. Encode a list of known passwords for use in decrypting data bytes.
3. For each known password (until finished or positive result):
    a. Decrypt the data length header (four bytes).

b. Check length against number of available bytes. If too large, go to next password.
c. Decrypt the rest of the data and retrieve stored checksum (one byte).
d. Calculate fresh checksum using decrypted data bytes.
e. Compare checksums.
    i. If checksums are equal, return True.
    ii. Else, go to next password.

4. Return False.

Extracting the eligible cover bytes in spdec.py means first filtering the MCUs according to SteganPEG's specifications, detailed above. Then, a function called dataBytes() reconstitutes the hidden bytes from the LSBs, tailoring itself to SteganPEG's embedding system (see Figure 15).

```
56  #Reconstruct data bytes from LSB's
57  def dataBytes(bList):
58      byteProcd = 0
59      res = []
60      count = 1
61      for b in bList:
62          bit = b % 2
63          if (b < 0):
64              bit = (~bit) & 1 #Reverse bit if negative
65          byteProcd = (byteProcd << 1) + bit
66          if (count == 8):
67              res.append(byteProcd)
68              byteProcd = 0
69              count = 0
70          count += 1
71
72      return res
```

Figure 15: Function dataBytes() from spdec.py

The function encodePass() is essentially a Python transcription of SteganPEG's original routine. This is used in spdec.py to generate a list of known, encoded passwords that will be needed to attempt decryption. As an initial proof of concept, the known list of passwords used in this iteration of the artifact is the top 20 most common passwords used in 2020, according to NordPass (List of the most common passwords, n.d.).

Next, each of the encoded passwords is used as a decryption key when trying to identify data as having been processed by SteganPEG. Adapted from the Visual Basic source

code, the function decryptData() takes as arguments a list of bytes to decrypt (byteList), a decryption key (passStore), and a special index number (rotChosen). Part of this decryption process also involves a bit rotation function called rotateLeft() (see Figure 16).

```
104 #RotateLeft (part of decryption)
105 def rotateLeft(val):
106     msb = val >> 7
107     newVal = val << 1
108     val = newVal & 0xFF
109     val = val | msb
110     return val
111 
112 
113 #General decryption sequence, returns a list of decrypted data bytes
114 def decryptData(byteList, passStore, rotChosen):
115     i = 0
116     result = []
117     while (i < len(byteList)):
118         val = byteList[i]
119         if (rotChosen == len(passStore)):
120             rotChosen = 0
121         for j in range(0,passStore[rotChosen]):
122             val = rotateLeft(val)
123         rotChosen += 1
124         result.append(val)
125         i += 1
126 
127     return result
```

Figure 16: Main decryption code from spdec.py

Since SteganPEG prepends the hidden data with four bytes that store the data length, those four bytes are decrypted first and the length value is compared with the total number of available possible bytes. If the decrypted length value is greater than the number of available bytes, it can be concluded that a match does not exist for SteganPEG using the current password and analysis should move to the next password. The section of code in spdec.py that handles this task is part of the entrypoint function detect() (see Figure 17).

```
170         #Decrypt first four data bytes, extract length of main embedded data
171         header = decryptData(bList[:4], passStores[i], 0)
172         dataLen = 0
173         for b in header:
174             dataLen <<= 8
175             dataLen = dataLen | b
176 
177         #If decoded length is more than (bList-4), data size too big, go to next pw
178         if (dataLen > (len(bList) - 4)):
179             msgs += "Data length too big, negative match...\n"
180             continue
```

Figure 17: Checking data length in function detect() from spdec.py

If the data length does not disqualify the sample, the rest of the data is decrypted and the last byte of the data is popped from the list, assumed to be the stored checksum. Finally, the function calcChecksum(), another routine adapted from the original source code, calculates a fresh checksum using the decrypted data (see Figure 18). Then, the last task in the detect() function is to compare both checksum values, returning true if they are equal. If no positive match is found, the detector will continue until all known passwords have been checked (see Figure 19).

```
130 #Calculate checksum for verification
131 def calcChecksum(data): #Pass in embed data without 4-byte length header and checksum
132     checksum = 0
133     sum = 0
134     val = 0
135     for i in range(len(data)):
136         val = (val << 1) | (data[i] & 1)
137         if (i % 7 == 0):
138             sum += val
139             val = 0
140     if (val != 0):
141         sum += val
142         val = 0
143     for j in range(8):
144         checksum = (checksum << 1) | (sum & 1)
145         sum >>= 1
146 
147     return checksum
```

Figure 18: Checksum calculation routine from spdec.py

```
185         decData = decryptData(bList2, passStores[i], rot)
186         checksum1 = decData.pop()
187 
188         #Validate checksum
189         checksum2 = calcChecksum(decData)
190         if (checksum1 == checksum2):
191             msgs += "Positive match!\n"
192             return True, msgs
193 
194         msgs += "Negative match...\n"
195 
196     return False, msgs
```

Figure 19: Comparing checksums in function detect() from spdec.py

OutGuess and ogdec.py

A negative result from both jsdec.py and spdec.py triggers the final detection module, ogdec.py, written to detect OutGuess. The algorithm used by OutGuess introduces complexities not found in either Jsteg or SteganPEG, and the program leaves only a minimal type of signature that makes it difficult to say conclusively that it has modified a JPEG, although in some cases it can be eliminated as a possibility. Based on these observations, it is shown below that the ogdec.py module uses two special values to implement a type of heuristic detection strategy for OutGuess.

OutGuess (Filho, 2019) is written in the C language and the source code makes use of many main component files and JPEG library modules, some of which are altered with custom code. It is a command-line tool that only runs on Linux systems. The program allows for an optional password, or it uses a default key value if one is not supplied. In addition, OutGuess implements its own ARC4 pseudorandom number generator which it uses for both encryption and for an iterator that chooses the bytes it uses for hiding data. ARC4 is a stream cipher that "generates a pseudorandom stream of bits" (RC4, n.d.). In order to be able to select these pseudorandom locations for steganography, a "bitmap" structure is created that extracts the eligible bytes from the cover image and stores them in an array. Statistical foiling options are also available, but they have minimal relevance to the detection methods in this study and are not considered here.

OutGuess uses two different ARC4 pseudorandom number streams in its algorithm. One is used as the iterator that chooses which cover bytes will contain the hidden message bits, and the other is used to encrypt the data by doing an XOR operation with the message bytes. The password for a given encoding session is used to initialize both of these streams, ensuring that OutGuess can find and decrypt the correct message during retrieval if the same password is supplied. It is also essential to note that before encryption, a special header is prepended to the main message data that contains the length of the message data, as well as a "seed" number that is used to reseed the iterator for when the main body of the message is retrieved. This prepending of information is reminiscent of both Jsteg's and SteganPEG's algorithms, and those two "signature" values play a key role in JRevealPEG's detection strategy for OutGuess.

It was explained above how OutGuess selects cover bytes from the bitmap structure, as well as how the data is encrypted. However, it is also essential to understand how the bitmap itself is constructed, specifically which MCU bytes are selected and in what order. Careful analysis of the source code shows that OutGuess makes use of all components (Y, Cb, and Cr), including all coefficients, DC and AC. The values 0 and 1 are avoided. In addition, three crucial points are observed. First, note that each 8x8 MCU block returned by coefx.py is represented as a one-dimensional list in so-called "zigzag" order, as opposed to natural order (see Figure 20), which agrees with the JPEG standard (JPEG, n.d.). This is not an issue when it comes to Jsteg and SteganPEG analysis, as the data in both of those programs is handled in zigzag ordering. However, the bitmap structure in OutGuess stores the selected MCU bytes in natural ordering.

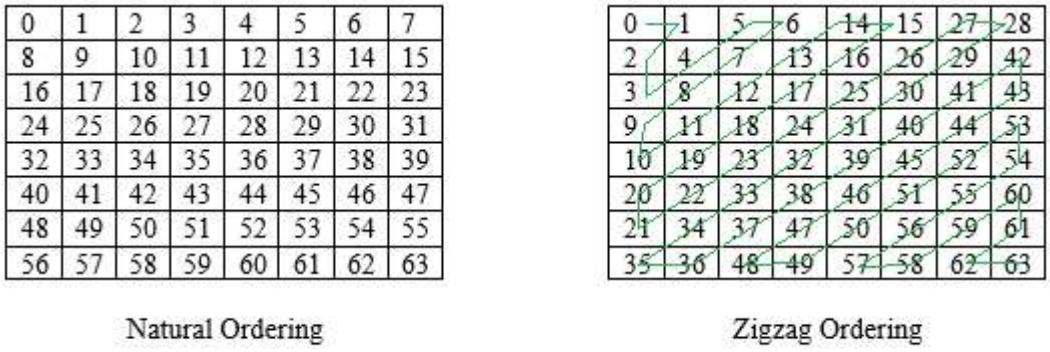

Figure 20: MCU Block – Natural vs Zigzag Ordering

The second critical point has to do with JPEGs that have visual dimensions not evenly divisible by 16. Recall that for these irregularly-sized JPEGs, the MCUs are padded so that the stored version of the image is indeed divisible by 16, even though this padding is discarded when the image is displayed. When OutGuess builds its steganography bitmap, the padding bytes are not included. This is an important consideration for the detection algorithm, which needs to rebuild this bitmap. Finally, the third crucial point to note is that OutGuess's embedding method results in even-valued bytes that decode to 0-bits, while odd values decode to 1-bits.

The ogdec.py module relies on byte analysis combined with special heuristics to determine if a JPEG could possibly contain OutGuess steganography, or if it can be

disqualified as a candidate. The decision to limit detection to only a "possible" positive was based on the analysis described above, in which the signature identified was not necessarily a conclusive fingerprint of OutGuess, but more of an indication of likelihood. Also, for proof-of-concept purposes, it is assumed that default settings were used for the steganography, including the default password. The overall detection strategy of ogdec.py can be described by the following steps:

1. Construct OutGuess-style bitmap list from JPEG MCUs (coefx.py output):
    a. Put the coefficients in natural order.
    b. Trim edge padding if necessary.
    c. Filter out 1's and 0's.
    d. Extract and save list of LSBs.
2. Extract seed value and data length from bitmap.
    a. Use precalculated iterator and encryption key values.
    b. Assume default password: "Default key".
3. Run heuristic checks:
    a. If data length is larger than half the size of the bitmap list, return False.
    b. If seed value is greater than 255, return False.
4. If both heuristic checks pass, return True (OutGuess possible).

The initial task of constructing an OutGuess-style bitmap list must be exact and is the most involved process in ogdec.py, containing several subtasks. The first subtask of converting the MCU blocks from zigzag to natural ordering occurs in dezig(), a fairly straightforward function that re-maps the indexes of each block (see Figure 21).

```
#Put coeff matrices in natural order
def dezig(coeffs):

    natOrder = [0, 1, 8, 16, 9, 2, 3, 10,
                17, 24, 32, 25, 18, 11, 4, 5,
                12, 19, 26, 33, 40, 48, 41, 34,
                27, 20, 13, 6, 7, 14, 21, 28,
                35, 42, 49, 56, 57, 50, 43, 36,
                29, 22, 15, 23, 30, 37, 44, 51,
                58, 59, 52, 45, 38, 31, 39, 46,
                53, 60, 61, 54, 47, 55, 62, 63]

    dezigged = []
    for c in coeffs:
        output = [0] * 64
        for i in range(64):
            output[natOrder[i]] = int(c[i])
        dezigged.append(output)

    return dezigged
```

Figure 21: Function dezig() from ogdec.py

The second subtask of bitmap construction is to remove any padding bytes that may be present in the MCU blocks of the right and bottom edges of the image matrix. For the JPEGs accepted by the detector, this routine only needs to occur if either the height or the width dimension is not divisible by 16. In ogdec.py, this process occurs in two functions: calcEdges() and trimEdges(). First, the function calcEdges() determines the indexes of the MCUs on the right and/or bottom edge, whichever has indivisible dimensions (modulo 16). A list of lists containing these indexes is returned for use in trimEdges() (see Figure 22). Second, the function trimEdges() must remove the padding bytes from the MCUs identified by calcEdges(). It was determined through testing that removing only the extra bytes in the appropriate Y components was sufficient in this case. However, the Cb and Cr components may need to be considered in future iterations of the artifact. An example of how trimEdges() calculates and removes the appropriate coefficients is shown in Figure 23. For a more in-depth understanding of how the Y components in 2x2 subsampling are mapped to a JPEG's pixel display, Hass (2018) provides an excellent explanation.

```python
#Calculate indexes of edge MCUs (right, bottom, bottom-right corner)
def calcEdges(y, x): #y is image height, x is width

    #Account for divisible lengths
    if (x % 16 == 0):
        xEdge = x // 16 - 1
    else:
        xEdge = x // 16

    if (y % 16 == 0):
        yEdge = y // 16 - 1
    else:
        yEdge = y // 16


    #Do right side list
    rSide = []
    rStart = xEdge
    rStep = xEdge + 1
    rStop = rStep * yEdge
    if (x % 16 != 0):
        for i in range(rStart, rStop, rStep):
            rSide.append(i)

    #Do bottom list
    bSide = []
    bStart = (xEdge + 1) * yEdge
    bStop = bStart + xEdge
    if (y % 16 != 0):
        for i in range(bStart, bStop):
            bSide.append(i)

    #Do bottom-right corner
    brCorner = [bStart + rStart]

    incomps = [rSide, bSide, brCorner]

    return incomps
```

Figure 22: Function calcEdges() from ogdec.py

```python
    #Edit rSide if necessary
    xRem = x % 16
    if (xRem != 0):
        if (xRem <= 8):
            off = 0
            pSave = xRem
            brY1 = brY3 = False
        else:
            off = 1
            pSave = xRem - 8

        #Calc pixels to keep
        leftKeep = []
        for row in range(0, 57, 8):
            for col in range(row, pSave + row):
                leftKeep.append(col)

        #Keep relevant pixels
        for mcu in incomps[0]:
            ylistA = []
            ylistB = []

            for c in range(len(mcuList[mcu*6 + off])):
                if (c in leftKeep):
                    ylistA.append(mcuList[mcu*6 + off][c])
            mcuList[mcu*6 + off] = ylistA

            for c in range(len(mcuList[mcu*6 + off + 2])):
                if (c in leftKeep):
                    ylistB.append(mcuList[mcu*6 + off + 2][c])
            mcuList[mcu*6 + off + 2] = ylistB
```

Figure 23: Excerpt of function trimEdges() from ogdec.py

The last two subtasks of bitmap construction in ogdec.py are much more lightweight than the first two. The 1s and 0s are filtered from the byte collection in much the same way as in previous modules. The final bitmap list is then populated with the LSBs from the remaining bytes by the bitmap() function, which decodes 0s from even bytes and 1s from odd bytes.

Once the bitmap list has been prepared, ogdec.py looks for special header values (two bytes each) that would have been embedded with the message data if the sample was indeed processed by OutGuess. These values include a "seed" number used in OutGuess's algorithm, and the length of the embedded message data. For the purposes of this study, it is assumed that only default options were used in any OutGuess-modified JPEGs. This includes a default password, which OutGuess sets to be "Default key".

The password is used to initialize both the iterator that chooses the sequence of cover bytes used, and the encryption key values. Fortunately, since it is assumed every sample only uses the default password, it was not necessary try to adapt to Python all the C routines from the source code that would replicate these values dynamically. Instead, the needed numbers were extracted while executing a session of OutGuess through a debugger, and they have been hardcoded in ogdec.py. The function extractHeaderInfo() first uses the 32 saved iterator values to find the correct bits in the bitmap list, which are then used to assemble the four supposed header bytes. The header bytes are each decrypted by an XOR operation with the saved encryption key values. The first two header bytes become the seed value, and the last two bytes become the data length, and these are returned for the next step (see Figure 24).

```
#Extract random seed and data length from beginning of data
def extractHeaderInfo(data):

    #Pre-calculated iterator and encryption key values for "Default key"
    iterator = [29, 50, 70, 71, 86, 93, 125, 140,
            167, 172, 194, 209, 233, 238, 264, 294,
            308, 313, 326, 336, 362, 388, 402, 413,
            426, 452, 463, 495, 503, 512, 520, 526]

    arc4key = [0x3f, 0x5e, 0xd7, 0x1c]

    #Reconstruct 4 header bytes (encrypted)
    encBytes = []
    for i in range(7,32,8):
        byte = 0
        off = 7
        for j in range(i,i-8,-1):
            byte += data[iterator[j]] << off
            off -= 1
        encBytes.append(byte)

    #Decrypt header
    hdr = []
    for i in range(4):
        hdr.append(encBytes[i] ^ arc4key[i])

    #Calc seed and data length
    seed = (hdr[1] << 8) + hdr[0]
    dataLen = (hdr[3] << 8) + hdr[2]

    return seed, dataLen
```

Figure 24: Function extractHeaderInfo() from ogdec.py

The final task in the ogdec.py detection process is to perform two heuristic tests in order to decide whether or not to disqualify the sample as OutGuess. If it is not disqualified the detector will report the sample as a possible, but not conclusive match for the tool. These tests occur at the end of the detect() function in ogdec.py (see Figure 25). The first test checks the calculated data length. Analysis of Outguess shows that it does not embed a message that is more than half the size of the bitmap list. Therefore, if the data length calculated by the detector is larger than this value, it can rule out OutGuess for the sample and return False. If the first test is passed, the seed value is checked to see if it is larger than 255, which is the limit observed through OutGuess code analysis. If the seed is larger than the limit, the detector can rule out OutGuess for the sample and return False. If these tests do not eliminate OutGuess as a possibility, the detector concludes that it is a possibility, assuming default options were used.

```
328     #OutGuess sets this data size limit
329     if (dataLen > len(bmap) // 2):
330         msgs += "Data length too big vs. available cover..."
331         return False, msgs
332     #Seed should be 255 or less for default key
333     if (seed > 255):
334         msgs += "Seed too large..."
335         return False, msgs
336 
337     #If header doesn't rule it out, assume OutGuess possible
338     return True, msgs
```

Figure 25: Heuristics tests in function detect() from ogdec.py

Theory and Artifact Design: Summary

This chapter began by introducing the design problem and research questions associated with the development of JRevealPEG, the artifact that is the object of this study. The initial requirements of the artifact were stated, and the three steganography tools that are the targets of JRevealPEG were also introduced. The architecture and behavior of each of the five modules of JRevealPEG were then discussed, along with relevant analysis details regarding the target steganography programs. The next chapter discusses the design and results of a single-case mechanism experiment that was performed as validation for this research.

CHAPTER 4

EXPERIMENT RESULTS AND DISCUSSION

This chapter begins with a brief discussion of the phases of a single-case mechanism experiment, which is the chosen design validation for this research. Next, the setup and execution of the experiment itself is addressed under Experiment Setup and Execution. Finally, the results of the experiment and data reports are discussed in the Results and Discussion section which also includes the answers to the three knowledge questions.

Design Validation

One means of validating design research is with a single-case mechanism experiment. Weiringa (2014) explains that this type of validation enables the user to "expose the model to controlled stimuli and analyze in detail which mechanisms are responsible for the responses" (p. 64). This seems to apply well in the case of this study, particularly in terms of answering the first two research questions, which pertain to finding levels of accuracy regarding hidden data detection and tool identification.

A single-case mechanism experiment consists of several pieces, most of which are summarized below as they relate to this study. These pieces include context, research problem, object of study, treatment design, measurement design, inference design, execution, and data analysis (Weiringa, 2014). The execution phase is discussed under the Experiment Setup and Execution section, and the data analysis is included in the Results and Discussion section.

Context and Research Problem

In terms of the context and research problem, the conceptual framework and knowledge questions for this study have been defined in the Introduction and other previous sections, including the Literature Review. The relevant variables include the size and dimensions of the JPEG file input, the processing time required to achieve a result, the detected presence of hidden data (Boolean), and the identified steganography tool of origin

(Jsteg, SteganPEG, or OutGuess). The population related to this validation is all instances of the detection tool being used by security professionals to detect JPEG steganography.

Object of Study

As the object of study, the validation model utilized a Windows 10 operating system environment with Python 3 and the artifact (the detection program) installed. The JPEG samples used for input were developed from a selection of free-to-use images downloaded from online sources. Some of these JPEGs were used as control samples and had no steganography embedded, while the remaining samples were processed by the target tools mentioned above and contained hidden data; specific details of the JPEG samples are provided in the Experiment Setup and Execution section.

The data generated include the output of the detection program itself, including the sizes and dimensions of the JPEG input files, relevant processing time measurements, and positive and negative detection results. It was expected that the validation model should behave similarly to a real-world implementation, since the basic environment used was a standard Windows 10 setup. There may be random variables that affect the validity of the results, however, such as human error when using the software or unknown system settings and environmental factors.

Treatment Design

The treatment design in the context of the validation consisted of providing pregenerated JPEG images as input to the detection program artifact. Other than the operating system and related software mentioned above, no special instruments were needed. The researcher had full knowledge and control of the JPEG images being used as input, however in real-world conditions, the possibility of uncontrolled input exists with other unpredictable conditions.

Measurement Design

In terms of measurement design, the variables of interest include JPEG file size, measured in bytes; JPEG image height and width dimensions, measured in pixels; detection processing times, measured in seconds; and three Boolean variables to indicate the presence

or absence of steganography relating to each of the target tools. The sources of data from the experiment were generated by the execution of the artifact itself. No special measurement instruments were needed, and data was initially stored and analyzed using basic spreadsheet software.

Inference Design

The inference design for the data generated in this study included descriptive, abductive, and analogic inferences. Descriptive summaries include charts, tables, and graphs showing raw output as well analysis of results from the treatment process, e.g. scatterplots showing the relationship between key metrics, such as input file size and processing time, and tables containing raw results from artifact execution.

As far as abductive inferences, any significant aberrations or inconsistencies in results could be explained by malformed input, such as a corrupt image file, random interruptions in the input and output streams to and from the artifact, or other software bugs.

Finally, the main analogic inference for the study is that the experiment would easily generalize to real-world cases, meaning security researchers using the artifact to analyze JPEG images on similar operating systems should see similar results. In fact, since the program is more dependent on a correctly-functioning language interpreter (Python) than a particular operating system, it is likely that cases involving a variety of operating systems would behave similarly.

Experiment Setup and Execution

As described under Object of Study, the experiment used to validate this design research consisted of a group of JPEG samples being passed as input to the artifact of the study, the JRevealPEG program. The program generated as its output several measurements per input sample, and this data was saved in log files for further analysis. Before the experiment could be officially executed, however, an appropriate group of JPEG samples had to be collected and prepared.

Setup

The previous chapter discussed the fact that the initial iteration of JRevealPEG is only configured to analyze color JPEGs that have Baseline compression and 2x2 chroma subsampling, mainly due to the fact that two of the three target steganography programs only output JPEGs with these specifications. When searching for random JPEGs online, there is no easy way to predict if a sample will be a Baseline or a progressive scan image, or what the subsampling ratio will be. Fortunately, it was found that the Paint3D program that comes with Windows 10 automatically converts any JPEG opened and then re-saved within the application to a Baseline, 2x2 image. Paint3D can also be used to vary the dimensions of a JPEG if more variation is needed.

In order to test for both positive and negative results, the JPEG samples needed to include images with and without steganography embedding. In addition, to test JRevealPEG's speed and handling of odd image dimensions, images within a wide range of file sizes, dimensional proportions, and total square pixel area were selected. In terms of the hidden messages used, two different types of files were chosen of relatively small size so they would be compatible with all sizes of cover images and all three embedding programs (see Table 2). However, the files are big enough to simulate a dangerous amount of sensitive data or a malicious executable. The message files can be found in the Windows 10 System32 folder.

Filename	File Type	Size (Bytes)
msdxm.ocx	Active X Control	7168
security.dll	DLL	5632

Table 2: Hidden messages used in the experiment

Table 3 shows an excerpt from the master list of JPEGs prepared to serve as input for the experiment. Ten different cover images of varying sizes and proportions were chosen to contain the hidden data. Each cover image was used twice with each target steganography program, once to hide msdxm.ocx, and once to hide security.dll. This generated 20 positive samples for each target program, or 60 positive samples total. Finally, the ten original covers were added to the list with no hidden data, and ten more random JPEGs were chosen and added, also with no hidden data. Therefore, the final group contained 60 positive and 20 negative samples, for a total of 80 JPEGs.

Stego File	Size (Bytes)	Height	Width	Sq. Pixels	Cover Image	Message File	Stego Program	Password?
jsPumpkinsSec.jpg	914303	3239	2293	7427027	pumpkins.jpg	security.dll	Jsteg	N/A
jsSpidersMsd.jpg	419257	2108	2400	5059200	spiders.jpg	msdxm.ocx	Jsteg	N/A
jsSpidersSec.jpg	419279	2108	2400	5059200	spiders.jpg	security.dll	Jsteg	N/A
jsYellowMsd.jpg	256814	1200	1200	1440000	yellow.jpg	msdxm.ocx	Jsteg	N/A
jsYellowSec.jpg	256821	1200	1200	1440000	yellow.jpg	security.dll	Jsteg	N/A
spCowMsd.jpg	1276041	2667	4000	10668000	cow.jpg	msdxm.ocx	SteganPeg	123123
spCowSec.jpg	1276026	2667	4000	10668000	cow.jpg	security.dll	SteganPeg	senha
spCraterMsd.jpg	1072285	1703	2556	4352868	crater.jpg	msdxm.ocx	SteganPeg	12345
spCraterSec.jpg	1072279	1703	2556	4352868	crater.jpg	security.dll	SteganPeg	Million2
spForestMsd.jpg	729943	1280	1920	2457600	forest.jpg	msdxm.ocx	SteganPeg	111111
ogGreentreesMsd.jpg	2648358	4293	6440	27646920	greentrees.jpg	msdxm.ocx	OutGuess	"Default key"
ogGreentreesSec.jpg	2648133	4293	6440	27646920	greentrees.jpg	security.dll	OutGuess	"Default key"
ogHouseplantMsd.jpg	254165	1066	1599	1704534	houseplant.jpg	msdxm.ocx	OutGuess	"Default key"
ogHouseplantSec.jpg	253888	1066	1599	1704534	houseplant.jpg	security.dll	OutGuess	"Default key"
ogPalmsMsd.jpg	216598	1200	1600	1920000	palms.jpg	msdxm.ocx	OutGuess	"Default key"

Table 3: Master list of JPEG samples (excerpt)

Also note that in Table 3 a password value exists where applicable with SteganPEG and OutGuess. With OutGuess the password is the same default for every sample, “Default key.” However, in order to test the full list of 20 known passwords included with the SteganPEG algorithm, every sample processed with that tool used a different password from that list.

Finally, the sample JPEGs were stored in a dedicated folder, the path of which would be passed to JRevealPEG during runtime. To facilitate command-line execution, the path to Python 3 was added to the Windows system environment variables, and a command window was opened to the directory containing the JRevealPEG scripts.

Execution

Once the group of JPEG input samples had been compiled in an accessible directory, executing the experiment was simply a matter of running the jrpeg.py script on a Windows 10 machine and entering the path to the input files at the appropriate time (see Figure 26). Once the detection sequence started, it proceeded automatically, displaying output to the screen, while also saving results to both log files, text and CSV. The text file is essentially a copy of the results displayed to the console, and the CSV is a distilled version of important statistics. Figures 27 and 28 show excerpts from both of the log files generated by the experiment, and the full files can be found in Appendices A and B.

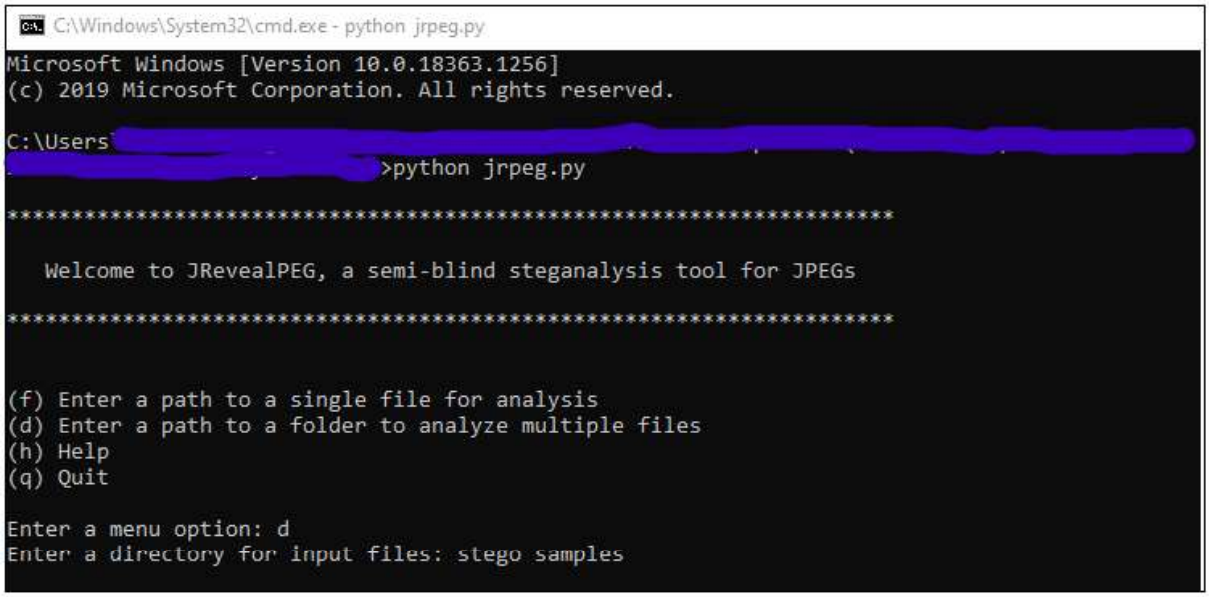

Figure 26: Experiment execution – running JRevealPEG with input samples

```
Extracting initial byte array from file 67, spPalmsSec.jpg. This could take several seconds...

Initial JPEG processing time: 3.07 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 67, spPalmsSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Positive match!


SteganPeg detected for file 67, spPalmsSec.jpg

Total detection and processing time for file 67, spPalmsSec.jpg: 3.95 seconds

Extracting initial byte array from file 68, spPenguinMsd.jpg. This could take several seconds...

Initial JPEG processing time: 9.44 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 68, spPenguinMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Positive match!


SteganPeg detected for file 68, spPenguinMsd.jpg

Total detection and processing time for file 68, spPenguinMsd.jpg: 11.70 seconds
```

Figure 27: Experiment execution - excerpt from JRevealPEG text log file

File	Size	Height	Width	Jsteg	SteganPEG	OutGuess	CoeffTime	OverallTime
cactus.jpg	591155	3600	2400	FALSE	FALSE	FALSE	4.97	11.23
cow.jpg	1275990	2667	4000	FALSE	FALSE	FALSE	10.33	18.54
crater.jpg	1072272	1703	2556	FALSE	FALSE	FALSE	8.52	12.4
eagle.jpg	770867	1634	2400	FALSE	FALSE	FALSE	5.97	9.23
forest.jpg	729925	1280	1920	FALSE	FALSE	FALSE	5.54	7.71
frog.jpg	298370	1805	2400	FALSE	FALSE	FALSE	2.49	5.56
greentrees.jpg	4664976	4293	6440	FALSE	FALSE	FALSE	38.45	62.32
houseplant.jpg	316713	1066	1599	FALSE	FALSE	FALSE	2.42	3.79
jsCowMsd.jpg	822003	2667	4000	TRUE	FALSE	FALSE	7.01	8.61
jsCowSec.jpg	822010	2667	4000	TRUE	FALSE	FALSE	7.41	8.96
jsCraterMsd.jpg	601331	1703	2556	TRUE	FALSE	FALSE	5.11	5.79
jsCraterSec.jpg	601336	1703	2556	TRUE	FALSE	FALSE	5.31	6.04
jsForestMsd.jpg	458084	1280	1920	TRUE	FALSE	FALSE	3.78	4.18
jsForestSec.jpg	458099	1280	1920	TRUE	FALSE	FALSE	3.76	4.16
jsGreentreesMsd.jpg	2650002	4293	6440	TRUE	FALSE	FALSE	23.06	27.24

Figure 28: Experiment execution – excerpt from JRevealPEG CSV log file

The experiment ran uninterrupted for approximately 15.7 minutes until completion. The next section analyzes the specific data results, as well as providing other observations of interest. Finally, there is a discussion of the answers to the three knowledge questions.

Results and Discussion

The following discussion begins with a selective analysis of the data generated by this experiment, in terms of the variables of interest described under Measurement Design, as well as any other interesting observations. Then, based upon these results and notes from the development period, a discussion is provided that answers the three original knowledge questions relating to this research.

Data Analysis

Since the relevant metrics of interest produced by the experiment were saved in a CSV log file, most of the data analysis related to these results could be derived using Microsoft Excel. When considering the kinds of data analyses that would best align with the current study's research goals, three useful perspectives of interest were identified: selected descriptive statistics derived from the main numeric metrics, correlation analysis between input file parameters and processing times, and accuracy of the Boolean detection results. Basic descriptive statistics can serve multiple purposes, such as providing a compact summary

of experiment input parameters and artifact performance, or setting a baseline for future experiments. Analysis of possible correlations between input samples and processing times exhibited by the artifact can facilitate predictive formulas, which could help judge the usefulness of the artifact in other experiments. Finally, calculating the accuracy of the Boolean detection results not only evaluates the effectiveness of the most important function of the artifact, but also provides the data necessary to answer the first two knowledge questions.

In terms of descriptive statistics, the raw output metrics considered included the size of the input file in bytes, the coefficients-only processing time for each file in seconds (coefx.py processing time), and the overall processing time for each file in seconds, including detection processing. In addition, the derived metrics of area in square pixels (image height times the width) and detection processing time (overall time minus coefficient time) were analyzed. The descriptive statistics considered the most relevant to this experiment and data set included, for each metric, the minimum and maximum, the range, and the mean. Table 4 provides a summary of these metrics in terms of the measured statistics.

Metric (per Input Sample)	Mean	Minimum	Maximum	Range
File Size (B)	949073	216003	4664980	4448977
Area in Sq. Pixels	6876053	1440000	27646920	26206920
Coefficient Processing Time	8.01	1.78	39.92	38.14
Detection Processing Time	3.77	0.22	23.87	23.65
Overall Processing Time	11.78	2.08	62.32	60.24

Table 4: Descriptive statistics for selected experiment metrics

The data in Table 4 shows that for the collection of input samples used in this experiment, which had an average file size of about 950KB, the average overall processing time for each file was close to 12 seconds. For 80 samples, this would mean a total execution time of 16 minutes, which was approximately the full runtime of the experiment. Some files only took about two seconds to process, while others took more than a minute. It can also be observed that the coefficient processing time took more than twice as long as detection processing on average and was about two-thirds of the overall processing time. The square-pixel area values are also shown as an alternative JPEG size measurement, but from this table there is no way of knowing if any specific correlations exist between size and area, or between other pairs of metrics. Finally, the percentages of positive and negative

steganography samples used in the experiment are not shown here, nor are the relative percentages of samples processed by each target tool. However, a good overall snapshot of the experiment and processing performance of JRevealPEG can be assessed.

Several interesting observations were made when analyzing possible correlations between certain input file characteristics and the different processing time metrics. The following pairs of metrics were analyzed as scatterplots with linear regression:

1. File Size vs Coefficient Processing Time
2. File Size vs Detection Time
3. File Size vs Overall Processing Time
4. Pixel Area vs Coefficient Processing Time
5. Pixel Area vs Detection Time
6. Pixel Area vs Overall Processing Time

Figure 29 shows the three graphs with file size as the horizontal axis, and Figure 30 shows the three graphs with square-pixel area as the horizontal axis. Note that each scatterplot has been superimposed with a regression line, the line of best fit for the given data. Also included are the equations for each line, and the R-squared values (recall that the closer R-squared is to 1, the stronger the correlation). If one examines the R-squared values first, it is apparent that the strongest correlation occurs between the input file size and the coefficient processing time, with a value of 0.9978. This value is not only the largest when compared with the other R-squared values, but it is extremely close to 1. In terms of coefficient processing time in future experiments, this would suggest that the file size of a particular input JPEG may be the best predictor of that value. The file size does not seem quite as good at predicting the overall processing time, with an R-squared of 0.9231, although still relatively close to 1. The pixel area has a weaker relationship with coefficient and overall processing time than file size does, but it does seem to carry more weight in determining the overall time (R-squared being 0.8423). Another interesting observation is that neither file size nor pixel area have a particularly strong correlation with the detection processing time metric, both having relatively mediocre R-squared values (0.5333 and 0.6102). This suggests that other factors exist which likely weigh in with importance, such as the relative presence or absence of steganography in a set of samples, as well as the proportion of samples embedded with data by a particular steganography tool.

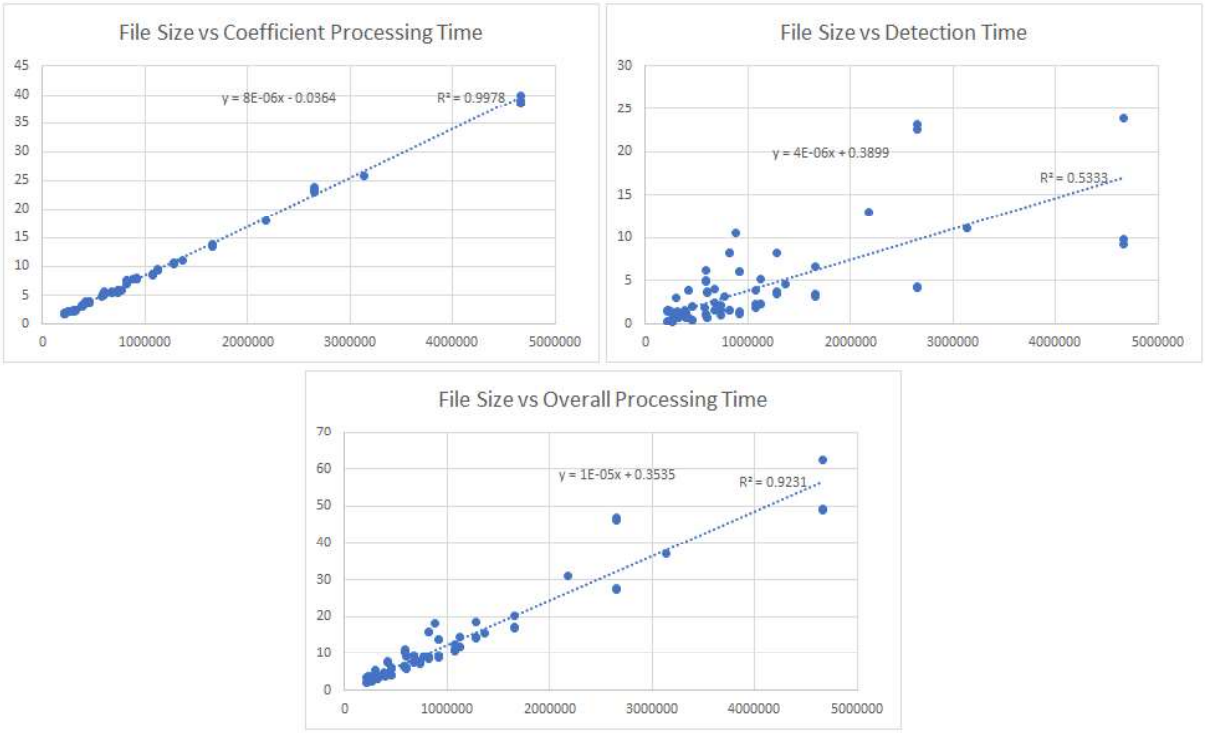

Figure 29: Graphs of file size vs processing times

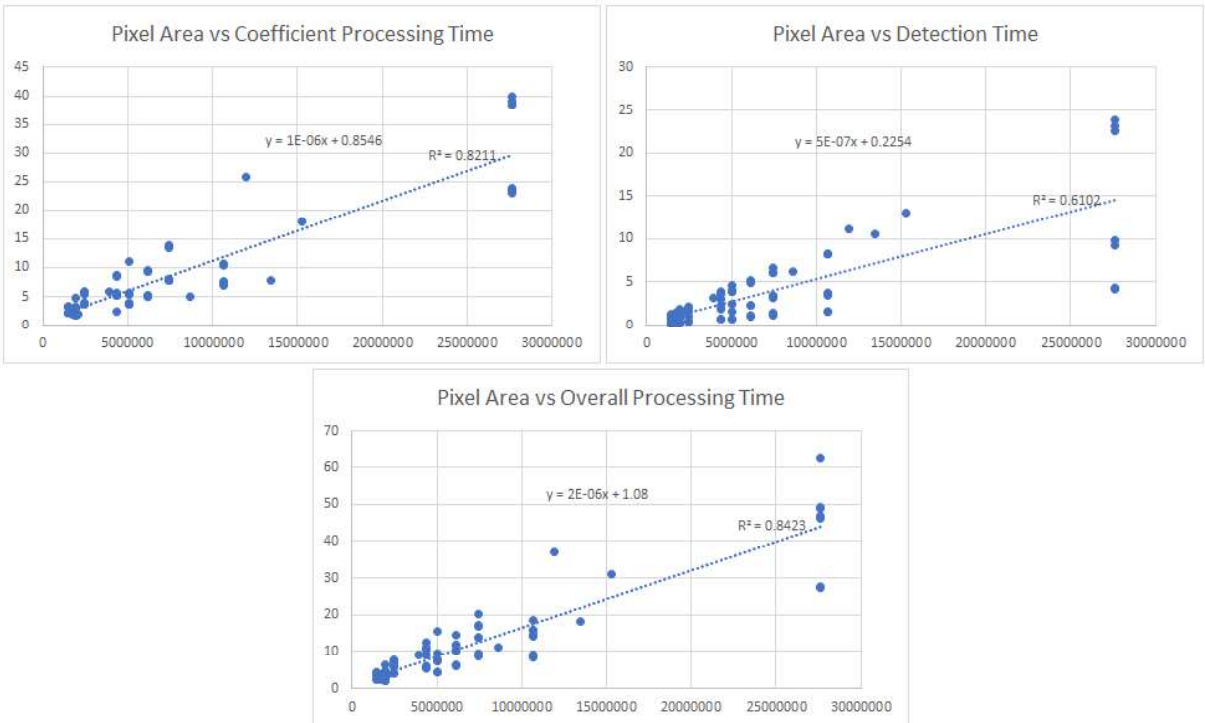

Figure 30: Graphs of square-pixel area vs processing times

Since the main purpose of JRevealPEG as a software tool is to detect steganography, the accuracy of its results when applied to a particular JPEG sample set is of paramount importance when judging its success. There are three Boolean variables that account for detection accuracy in JRevealPEG, one for each target steganography program and labeled with the name of each tool (Jsteg, SteganPEG, and OutGuess). According to the CSV report generated as part of the output of the experiment, JRevealPEG was correct in its classification of all 80 samples (see Appendix B for the full CSV report). The filenames used for each JPEG can be used to easily verify each result in the CSV file: JPEGs positive for Jsteg begin with "js" followed by a capital letter; those of SteganPEG have "sp" followed by a capital; OutGuess files have "og" followed by a capital; the remaining files should be negative for any steganography.

Of course, future iterations of the experiment with key factors adjusted to greater extremes, such as size and variety of message payloads and of cover images, may expose errors or inaccuracies not discovered during the execution of the experiment at the time of this study. Additionally, as noted during the discussion of JRevealPEG's architecture, there were certain assumptions made to facilitate what is really a "conditional" detection capability for SteganPEG and OutGuess, such as use of known passwords. Future attempts to eliminate these assumptions may reduce the level of accuracy of which the artifact currently appears to be capable.

Answers to Knowledge Questions

The original knowledge questions raised in Chapter 3 of this study are restated as follows:

1. Within the scope of the embedding tools targeted by this study, what level of accuracy can be achieved by the program in terms of successfully detecting the presence of steganography in a given JPEG?
2. If steganography is detected in a JPEG, what level of accuracy can be achieved by the program in terms of successfully identifying which target tool was used to hide the data?

3. What kinds of obstacles and difficulties were encountered in terms of designing successful detection methods for the steganographic tools and embedding methods targeted by this study, and which (if any) of these obstacles were not overcome?

Answer to Question 1: According to the results of the experiment, out of 60 positive samples and 20 negative samples, JRevealPEG detected the presence of steganography with 100% accuracy, i.e. no false positives and no false negatives occurred.

Answer to Question 2: According to the results of the experiment, out of the 60 samples it classified as positive for steganography, JRevealPEG successfully identified the target tool used with 100% accuracy, i.e. there were no false attributions for any of the detected steganography.

Answer to Question 3: The difficulties and obstacles encountered during the research and design of the JRevealPEG artifact were many, varied, persistent, and often unexpected. Fortunately, in virtually all cases a direct solution or workaround was found, which allowed the research to progress and an artifact to be developed that was able to address the original research goals. Most of the biggest obstacles arose during JPEG processing, and while reverse engineering the target steganography programs.

JPEG Processing: One of the main areas that spawned several challenges early in the research period was JPEG processing in Python 3. Initially, it appeared that there may be one or more existing Python libraries that contained JPEG processing functionality which could be used in this project. Unfortunately, every library that was examined contained functions that were either too "blunt" for the granularity needed in this artifact, or the functionality needed was embedded deep within other, more global routines. The main problem was most programs served to fully decompress JPEG images; the artifact in this study needed to extract data from only partway through the decompression process. Eventually a single script was found (mentioned in Chapter 3) that seemed to contain functionality close to that which was needed for this research, or at least it seemed like it could be adapted.

Now that a Python module was found that could be used as a starting point to aid in JPEG processing, the challenge arose of adapting the code to the specific needs of this

artifact. This challenge had three main difficulties. First, the finer details of JPEG compression are not trivial, and much related research had to be done to get up to speed with sufficient expertise. Second, the only sure way to know if the extracted partially-decompressed data was correct (the pre-Huffman scan bytes) was to allow the decompression to go to completion during time-consuming debugging, then verify that the final image was displayed correctly. Third, the script that was being adapted was not compatible with 2x2 subsampling, so much learning was needed and many modifications had to be made to fix this. In the end, adapting that Python script to extract the needed data was a major milestone in the study.

Reversing the Target Programs: Another essential task that presented a wide variety of challenges was the effort to reverse engineer the three target steganography programs. This was a necessary step toward understanding the tools on a deep-enough level to be able to identify possible steganography signatures, invaluable for the development of the artifact's detection algorithms. Some difficulties applied to all three programs, others were tool-specific.

One problem that initially became apparent was that each tool was written in a different programming language (Go, Visual Basic, and C). This was not unexpected, but it took time to become accustomed to the various syntax and data type differences of each. Occasionally mistakes resulted from transcriptional errors between a target's source language and Python 3. The disparity of programming languages and architectures also made tracing each target program an adventure. Tracing OutGuess, in particular, involved going back and forth through dozens of .h and .c source files.

The target programs also had varying levels of documentation, and some of the comments in the source code were rather cryptic. In terms of message-hiding capacity, it was not clear initially how much data each program could hide in a cover image. Only through experimentation and tracing were limitations in this area determined.

As explained in Chapter 3, LSB steganography in a JPEG occurs in the partially-compressed image bytes. A big challenge in this research was to be able to reconstruct exactly which of those partially-compressed bytes each of the target programs chose when hiding data. Jsteg was the most straightforward, in that the program essentially hid data in consecutive bytes from those that were eligible. For SteganPEG, the main difficulty was

recognizing that the message data was compressed before it was hidden, which threw off size comparisons during steganography analysis. The last program to be cracked in terms of its chosen bytes was OutGuess. First, it took a while to realize that OutGuess stores the chosen cover bytes in natural order, which is different than the zigzag order used by Jsteg and SteganPEG. Then, much analysis was required to discover the special manner in which OutGuess handles JPEGs with uneven dimensions (not divisible by 16), namely that it ignores padding bytes.

Experiment Results and Discussion: Summary

This chapter defined the components of a single-case mechanism experiment, which is the form of treatment validation used for this design science research. It explained the setup and execution of the experiment involving the artifact of the study, JRevealPEG. The results of the experiment were then discussed in terms of analysis of the data generated. Finally, answers to the three original knowledge questions were provided. The final chapter is the Conclusion, which gives a summary of this paper, lists the expected scholarly contributions of this research, identifies the primary limitations of the artifact developed in this study, and recommends several possible future research directions related to this work.

CHAPTER 5

CONCLUSION

This work introduced the design of JRevealPEG, a steganalysis software tool written in Python 3 that targets three popular, open-source steganography programs. A brief history of steganography was provided, along with reasoning behind the motivation for this study and the current significance, interest, and need for a steganalysis tool like JRevealPEG. Several examples from current literature were presented, illustrating recent efforts in JPEG steganalysis research, including techniques such as those involving machine learning. The architecture and behavior of JRevealPEG was examined in relation to each of its five Python modules, and its detection algorithms were discussed along with the steganography programs that they target. Finally, the results of a single-case mechanism experiment used as treatment validation were analyzed, and the three knowledge questions posed at the beginning of the research were answered.

The rest of this chapter identifies possible scholarly contributions and applications of this research, discusses the primary limitations of the artifact developed in this study, and recommends a number of future research directions related to this project.

Contributions and Applications

The original motivation for this research was to contribute to the field of cyber security as it relates to malicious activities involving the secret transmission of digital material, specifically through the JPEG medium. It is intended that JRevealPEG will be made freely available to the professional and academic cyber security community as a tool for both research and practical applications.

One area of cyber security research that should directly benefit from this research is the study of JPEG steganography and steganalysis, and possibly that related to other types of images as well. JRevealPEG can provide insight into current tools used for both steganography and steganalysis, as its design research involved reverse engineering three open-source tools that are also available for anyone to study.

Another area of cyber security research to which this work contributes is that of Python security tool development. It is hoped that insight will be gained as to how to take advantage of Python in adapting code from other programming languages, as well as writing other kinds of detection software in Python.

Practical applications exist that could also benefit from the usage of JRevealPEG as a detection tool. JRevealPEG is already equipped with some validation and exception handling, so it should be robust enough to be exposed to multiple types of files. One possible application could be to run JRevealPEG on an entire directory of files as a preliminary sweep, just to see if any detection is triggered.

Another application can be for security personnel to use JRevealPEG to help analyze multiple data exfiltration incidents to see if they are related. It is possible that the program could play a role in linking a group of stego images to a specific embedding tool.

It is also possible that JRevealPEG can help identify the source of other types of malicious attacks. For example, JPEGs can be used for surreptitious delivery of malware executables, or they can be used to plant sensitive data on a certain party's machine, intending to frame them for possessing illegal or unauthorized material. If one of the target tools was used to hide the data, it could be specifically detected.

Limitations of JRevealPEG

Although JRevealPEG was shown to function well when operating on input files within the scope of this study and falling within certain assumptions and parameters, there are several limitations on the functionality and usefulness of the program in its current version.

The most obvious shortcoming of JRevealPEG is that it is only tailored to detect steganography from the three target tools used in the study. Therefore, JPEGs carrying messages hidden by a program other than those three will likely go undetected. It is possible steganographic algorithms exist which are similar enough to one of the current targets that they might be detectable by JRevealPEG, but the likelihood of that is not known.

Another limitation is that JRevealPEG only analyzes Baseline JPEGs with 2x2 chroma subsampling. But in fact, this is really only a limitation as it relates to one of the target tools, SteganPEG, as the other two tools only output Baseline, 2x2 JPEGs in any case. However,

any image processed by SteganPEG (or any JPEG at all) that is not 2x2 subsampled would not be detectable. JPEGs that are encoded as progressive scan would also be excluded.

Even considering compatible JPEGs processed by one of the target tools, there are limitations on JRevealPEG's detection capability. As seen in the experiment from Chapter 4, when scanning a file under the SteganPEG detection algorithm, steganography will only be detected if it was embedded with one of a group of known passwords. This means a stego image that has used a password not on the list will escape detection. Similarly, the OutGuess detector only works for images that utilize the default options, which include a password of "Default key ." Other configurations are also available with OutGuess that could affect JRevealPEG's detection ability.

One other possible limitation to note has to do with multiple payloads. Both SteganPEG and OutGuess have the option of hiding multiple messages in a JPEG. After preliminary analysis of these tools, it remains unclear if the current detection algorithms of JRevealPEG would be able to detect an image processed by these tools that contains more than one hidden message. Some indications show that it might, but this contingency was not in the scope of this study and has not been tested.

Future Research Directions

The work performed in designing JRevealPEG raises many opportunities for future research in connection with this study. Some of these ideas are motivated by the desire to address the limitations mentioned in the previous section, and others relate to exploring the possibility of extending JRevealPEG's other functionalities beyond that which was originally intended.

Adding detection functionality for other embedding tools would address the first limitation mentioned above. Naturally one would start with tools that could be reverse engineered most easily, probably programs where the source code is available. A detection algorithm could conceivably be developed for a program without having the source code, but an extra layer of difficulty would be present when trying to reverse that tool.

To address the fact that JRevealPEG can only process JPEGs with 2x2 subsampling, more decoding functionality would have to be added to coefx.py that takes this into consideration. Practically speaking, this would probably not be a difficult task, but a natural

upgrade for further iterations of the program, since it would increase the range of SteganPEG detection ability. The capability to process progressive scan JPEGs could also be added, but this would only be necessary if new detection capabilities for a new tool merited the upgrade.

The limitation requiring the use of known passwords with SteganPEG and OutGuess would be a natural one to seek to eliminate, so that fully-blind detection could be accomplished. In SteganPEG's case, an efficient manner would have to be found to check all valid password combinations until a valid checksum could be calculated. OutGuess would be a bit more complicated, however, because the PRNG would have to be replicated inside the detector, which could degrade performance

One idea for a completely new feature upgrade for JRevealPEG is adding payload extraction capability. In fact, much of the ground work has already been done for this since the program already knows how to extract the pre-Huffman MCUs. These are the bytes that contain the steganography. The journey to full payload extraction might not be much farther. However, on some level it might be easiest just to decode an image inside the original program, once it has been detected as the originator.

One final idea for an extended feature for JRevealPEG would be to add detection capability for image types other than JPEG. In fact, since steganography can be performed on any file, audio and video files could perhaps be added in the future as well.

Summary

This study detailed the rationale for and the design of JRevealPEG, a new steganalysis software tool that detects steganography in JPEGs and identifies the responsible embedding program out of a select open-source group. The study provided a brief background on digital image steganography and steganalysis, focusing on the JPEG standard and literature that illustrates the current state of JPEG and other image steganography detection tools and techniques. The literature revealed that most of the current detection tools and methods use machine learning and focus on general detection, while there is a lack of research and tools that are not based on machine learning. JRevealPEG was designed to explore this research gap and test the effectiveness of a tool designed without machine learning that has specific targets of detection.

The results of this study showed that a program can be written without machine learning that effectively detects JPEG steganography and identifies the tools of origin. The design process of JRevealPEG illustrated that Python can be a useful language for doing JPEG analysis, but also that not many existing Python libraries facilitate low-level JPEG manipulation, illustrating another opportunity for research. The outcome of the single-case mechanism experiment performed with JRevealPEG showed that the tool proved highly successful in a controlled environment with specific parameters and limitations, but plenty of room exists for overcoming those limitations and extending JRevealPEG's capabilities to make it a more universal tool.

REFERENCES

Abhiram, K. (2011). SteganPEG (Version 1.0) [Computer software]. Retrieved from https://www.softpedia.com/get/Security/Encrypting/SteganPEG.shtml

Boroumand, M., Mo, C., & Fridrich, J. (2019). Deep residual network for steganalysis of digital images. *IEEE Transactions on Information Forensics and Security, 14*(5), 1181-1193. https://doi.org/10.1109/TIFS.2018.2871749

Brunot, J. (2019). *The Increased Use of Steganography by Malware Creators to Obfuscate Their Malicious Code.* ProQuest Dissertations Publishing.

Burney, M. (2018). *The History of Steganography and the Threat Posed to the United States and the Rest of the International Community.* ProQuest Dissertations Publishing.

Butora, J., & Fridrich, J. (2020). Reverse JPEG compatibility attack. *IEEE Transactions on Information Forensics and Security, 15*, 1444-1454. https://doi.org/10.1109/TIFS.2019.2940904

Champine, L. (2018). Jsteg (Version 0.3.0) [Computer software]. Retrieved from https://github.com/lukechampine/jsteg/releases/tag/v0.3.0

Chroma subsampling. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Chroma_subsampling

Cisomag. (2019, March 26). *Hackers using steganography to spread Powload malware: Research*. https://cisomag.eccouncil.org/hackers-using-steganography-to-spread-powload-malware-research /

Das, S., & Dhara, B. (2018). An LSB based novel data hiding method using extended LBP. *Multimed Tools Appl, 77*(12), 15321-15351. https://doi.org/10.1007/s11042-017-5117-8

Denemark, T., Boroumand, M., & Fridrich, J. (2016). Steganalysis features for content-adaptive JPEG steganography. *IEEE Transactions on Information Forensics and Security, 11*(8), 1736-1746. https://doi.org/10.1109/TIFS.2016.2555281

Dunaway, G. (2019, March 21). *VeryMal strikes again with a new twist on its complex redirect attack.* AdMonsters. https://www.admonsters.com/verymal-strikes-new-twist-complex-redirect-attack/

Feng, G., Zhang, X., Ren, Y., Qian, Z., & Li, S. (2020). Diversity-based cascade filters for JPEG steganalysis. *IEEE Transactions on Circuits and Systems for Video Technology, 30*(2), 376-386. https://doi.org/10.1109/TCSVT.2019.2891778

Filho, J. (2019). OutGuess (Version 0.2.2) [Computer software]. Retrieved from https://github.com/resurrecting-open-source-projects/outguess/releases/tag/0.2.2

Gan, J., Liu, J., Luo, X., Yang, C., & Liu, F. (2018). Reliable steganalysis of HUGO steganography based on partially known plaintext. *Multimed Tools Appl, 77*(14), 18007-18027. https://doi.org/10.1007/s11042-017-5134-7

Go (programming language) (n.d.). Retrieved from https://en.wikipedia.org/wiki/Go_(programming_language)

Hass, C. (2018). Designing a JPEG decoder & source code [Web log post]. Retrieved from https://www.impulseadventure.com/photo/jpeg-decoder.html

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

Jia-Fa, M., Xin-Xin, N., Gang, X., Wei-Guo, S., & Na-Na, Z. (2016). A steganalysis method in the DCT domain. *Multimedia Tools and Applications, 75*(10), 5999-6019. https://doi.org/10.1007/s11042-015-2708-0

JPEG (n.d.). Retrieved from https://en.wikipedia.org/wiki/JPEG

JPEG Compression (n.d.). Retrieved from https://www.fileformat.info/mirror/egff/ch09_06.htm

Kafeine. (2016, July 28). Massive AdGholas malvertising campaigns use steganography and file whitelisting to hide in plain sight [Web log post]. Retrieved from https://www.proofpoint.com/us/threat-insight/post/massive-adgholas-malvertising-campaigns-use-steganography-and-file-whitelisting-to-hide-in-plain-sight

Khalid, M. (2019). Baseline JPEG Decoder [Computer software]. Retrieved from https://github.com/yasoob/Baseline-JPEG-Decoder

List of the most common passwords (n.d.). Retrieved from https://en.wikipedia.org/wiki/List_of_the_most_common_passwords

Lu, W., Li, R., Zeng, L., Chen, J., Huang, J., & Shi, Y.-Q. (2019). Binary image steganalysis based on histogram of structuring elements. *IEEE Transactions on Circuits and Systems for Video Technology*, 1-1. https://doi.org/10.1109/TCSVT.2019.2936028

Luo, X., Song, X., Li, X., Zhang, W., Lu, J., Yang, C., & Liu, F. (2016). Steganalysis of HUGO steganography based on parameter recognition of syndrome-trellis-codes. *Multimedia Tools and Applications, 75*(21), 13557-13583. https://doi.org/10.1007/s11042-015-2759-2

Malik, H., Subbalakshmi, K., & Chandramouli, R. (2016). Joint-channel modeling to attack QIM steganography. *Multimedia Tools and Applications, 75*(21), 13585-13611. https://doi.org/10.1007/s11042-015-3006-6

Malvertising (n.d.). Retrieved from https://en.wikipedia.org/wiki/Malvertising

Nouri, R., & Mansouri, A. (2017). Digital image steganalysis based on the reciprocal singular value curve. *Multimedia Tools and Applications, 76*(6), 8745-8756. https://doi.org/10.1007/s11042-016-3507-y

Qiao, T., Luo, X., Wu, T., Xu, M., & Qian, Z. (2019). Adaptive steganalysis based on statistical model of quantized DCT coefficients for JPEG images. *IEEE Transactions on Dependable and Secure Computing*, 1-1. https://doi.org/10.1109/TDSC.2019.2962672

Rabee, A., Mohamed, M., & Mahdy, Y. (2018). Blind JPEG steganalysis based on DCT coefficients differences. *Multimed Tools Appl, 77*(6), 7763-7777. https://doi.org/10.1007/s11042-017-4676-z

RC4 (n.d.). Retrieved from https://en.wikipedia.org/wiki/RC4

Sairam, T., & Boopathybagan, K. (2019). Computational intelligence-based steganalysis comparison for RCM-DWT and PVA-MOD methods. *Automatika, 60*(3), 293. https://doi.org/10.1080/00051144.2019.1579434

Serrano, J. (2019). Steganalysis: a study on the effectiveness of steganalysis tools. In J. Giordano & R. DeCarlo (Eds.): ProQuest Dissertations Publishing.

Shulmin, A., & Krylova, E. (2017). Steganography in contemporary cyberattacks. Retrieved from https://securelist.com/steganography-in-contemporary-cyberattacks/79276/

Song, X., Liu, F., Zhang, Z., Yang, C., Luo, X., & Chen, L. (2017). 2D Gabor filters-based steganalysis of content-adaptive JPEG steganography. *Multimed Tools Appl, 76*(24), 26391-26419. https://doi.org/10.1007/s11042-016-4157-9

Subhedar, M., & Mankar, V. (2018). Curvelet transform and cover selection for secure steganography. *Multimed Tools Appl, 77*(7), 8115-8138. https://doi.org/10.1007/s11042-017-4706-x

Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Berlin: Springer.

Xu, C., Liu, J., Gan, J., & Luo, X. (2018). Stego key recovery based on the optimal hypothesis test. *Multimed Tools Appl, 77*(14), 17973-17992. https://doi.org/10.1007/s11042-017-4878-4

Yang, Y., Kong, X., & Feng, C. (2018). Double-compressed JPEG images steganalysis with transferring feature. *Multimed Tools Appl, 77*(14), 17993-18005. https://doi.org/10.1007/s11042-018-5734-x

Vijayan, J. (2017). Steganography use on the rise among cyber espionage, cybercrime groups. Retrieved from https://www.darkreading.com/attacks-breaches/steganographyuse-on-the-rise-among-cyber-espionage-cybercrime-groups/d/d-id/1329569

Zielińska, E., Mazurczyk, W., & Szczypiorski, K. (2014). Trends in steganography. *Communications of the ACM, 57*(3), 86-95. https://doi.org/10.1145/2566590.2566610

APPENDIX A

LOG FILE 1: TEXT FILE

Logged on [redacted]

Directory entered: stego samples

Number of files to be analyzed: 80

Extracting initial byte array from file 1, cactus.jpg. This could take several seconds...

Initial JPEG processing time: 4.97 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 1, cactus.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...

Checking password '123456'... Data length too big, negative match...

Checking password '123456789'... Data length too big, negative match...

Checking password 'picture1'... Data length too big, negative match...

Checking password 'password'... Data length too big, negative match...

Checking password '12345678'... Data length too big, negative match...

Checking password '111111'... Data length too big, negative match...

Checking password '123123'... Data length too big, negative match...

Checking password '12345'... Data length too big, negative match...

Checking password '1234567890'... Data length too big, negative match...

Checking password 'senha'... Data length too big, negative match...

Checking password '1234567'... Data length too big, negative match...

Checking password 'qwerty'... Data length too big, negative match...

Checking password 'abc123'... Data length too big, negative match...

Checking password 'Million2'... Data length too big, negative match...

Checking password '000000'... Data length too big, negative match...

Checking password '1234'... Data length too big, negative match...

Checking password 'iloveyou'... Data length too big, negative match...

Checking password 'aaron431'... Data length too big, negative match...

Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 1, cactus.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 1, cactus.jpg

None of the target steganography programs detected for file cactus.jpg

Total detection and processing time for file 1, cactus.jpg: 11.23 seconds

Extracting initial byte array from file 2, cow.jpg. This could take several seconds...

Initial JPEG processing time: 10.33 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 2, cow.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...

Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 2, cow.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 2, cow.jpg

None of the target steganography programs detected for file cow.jpg

Total detection and processing time for file 2, cow.jpg: 18.54 seconds

Extracting initial byte array from file 3, crater.jpg. This could take several seconds...

Initial JPEG processing time: 8.52 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 3, crater.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...

Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 3, crater.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 3, crater.jpg

None of the target steganography programs detected for file crater.jpg

Total detection and processing time for file 3, crater.jpg: 12.40 seconds

Extracting initial byte array from file 4, eagle.jpg. This could take several seconds...

Initial JPEG processing time: 5.97 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 4, eagle.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...

Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 4, eagle.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 4, eagle.jpg

None of the target steganography programs detected for file eagle.jpg

Total detection and processing time for file 4, eagle.jpg: 9.23 seconds

Extracting initial byte array from file 5, forest.jpg. This could take several seconds...

Initial JPEG processing time: 5.54 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 5, forest.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 5, forest.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 5, forest.jpg

None of the target steganography programs detected for file forest.jpg

Total detection and processing time for file 5, forest.jpg: 7.71 seconds

Extracting initial byte array from file 6, frog.jpg. This could take several seconds...

Initial JPEG processing time: 2.49 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 6, frog.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 6, frog.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 6, frog.jpg

None of the target steganography programs detected for file frog.jpg

Total detection and processing time for file 6, frog.jpg: 5.56 seconds

Extracting initial byte array from file 7, greentrees.jpg. This could take several seconds...

Initial JPEG processing time: 38.45 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 7, greentrees.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 7, greentrees.jpg

*****Final pass - OutGuess: attempting to find signature...

Seed too large...

OutGuess (with default options) not possible for file 7, greentrees.jpg

None of the target steganography programs detected for file greentrees.jpg

Total detection and processing time for file 7, greentrees.jpg: 62.32 seconds

Extracting initial byte array from file 8, houseplant.jpg. This could take several seconds...

Initial JPEG processing time: 2.42 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 8, houseplant.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 8, houseplant.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 8, houseplant.jpg

None of the target steganography programs detected for file houseplant.jpg

Total detection and processing time for file 8, houseplant.jpg: 3.79 seconds

Extracting initial byte array from file 9, jsCowMsd.jpg. This could take several seconds...

Initial JPEG processing time: 7.01 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 9, jsCowMsd.jpg

Total detection and processing time for file 9, jsCowMsd.jpg: 8.61 seconds

Extracting initial byte array from file 10, jsCowSec.jpg. This could take several seconds...

Initial JPEG processing time: 7.41 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 10, jsCowSec.jpg

Total detection and processing time for file 10, jsCowSec.jpg: 8.96 seconds

Extracting initial byte array from file 11, jsCraterMsd.jpg. This could take several seconds...

Initial JPEG processing time: 5.11 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 11, jsCraterMsd.jpg

Total detection and processing time for file 11, jsCraterMsd.jpg: 5.79 seconds

Extracting initial byte array from file 12, jsCraterSec.jpg. This could take several seconds...

Initial JPEG processing time: 5.31 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 12, jsCraterSec.jpg

Total detection and processing time for file 12, jsCraterSec.jpg: 6.04 seconds

Extracting initial byte array from file 13, jsForestMsd.jpg. This could take several seconds...

Initial JPEG processing time: 3.78 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 13, jsForestMsd.jpg

Total detection and processing time for file 13, jsForestMsd.jpg: 4.18 seconds

Extracting initial byte array from file 14, jsForestSec.jpg. This could take several seconds...

Initial JPEG processing time: 3.76 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 14, jsForestSec.jpg

Total detection and processing time for file 14, jsForestSec.jpg: 4.16 seconds

Extracting initial byte array from file 15, jsGreentreesMsd.jpg. This could take several seconds...

Initial JPEG processing time: 23.06 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 15, jsGreentreesMsd.jpg

Total detection and processing time for file 15, jsGreentreesMsd.jpg: 27.24 seconds

Extracting initial byte array from file 16, jsGreentreesSec.jpg. This could take several seconds...

Initial JPEG processing time: 23.52 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 16, jsGreentreesSec.jpg

Total detection and processing time for file 16, jsGreentreesSec.jpg: 27.83 seconds

Extracting initial byte array from file 17, jsHouseplantMsd.jpg. This could take several seconds...

Initial JPEG processing time: 1.95 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 17, jsHouseplantMsd.jpg

Total detection and processing time for file 17, jsHouseplantMsd.jpg: 2.22 seconds

Extracting initial byte array from file 18, jsHouseplantSec.jpg. This could take several seconds...

Initial JPEG processing time: 2.20 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 18, jsHouseplantSec.jpg

Total detection and processing time for file 18, jsHouseplantSec.jpg: 2.46 seconds

Extracting initial byte array from file 19, jsPalmsMsd.jpg. This could take several seconds...

Initial JPEG processing time: 1.79 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 19, jsPalmsMsd.jpg

Total detection and processing time for file 19, jsPalmsMsd.jpg: 2.08 seconds

Extracting initial byte array from file 20, jsPalmsSec.jpg. This could take several seconds...

Initial JPEG processing time: 1.80 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 20, jsPalmsSec.jpg

Total detection and processing time for file 20, jsPalmsSec.jpg: 2.09 seconds

Extracting initial byte array from file 21, jsPenguinMsd.jpg. This could take several seconds...

Initial JPEG processing time: 5.03 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 21, jsPenguinMsd.jpg

Total detection and processing time for file 21, jsPenguinMsd.jpg: 6.04 seconds

Extracting initial byte array from file 22, jsPenguinSec.jpg. This could take several seconds...

Initial JPEG processing time: 5.18 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 22, jsPenguinSec.jpg

Total detection and processing time for file 22, jsPenguinSec.jpg: 6.28 seconds

Extracting initial byte array from file 23, jsPumpkinsMsd.jpg. This could take several seconds...

Initial JPEG processing time: 8.01 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 23, jsPumpkinsMsd.jpg

Total detection and processing time for file 23, jsPumpkinsMsd.jpg: 9.40 seconds

Extracting initial byte array from file 24, jsPumpkinsSec.jpg. This could take several seconds...

Initial JPEG processing time: 7.77 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 24, jsPumpkinsSec.jpg

Total detection and processing time for file 24, jsPumpkinsSec.jpg: 8.93 seconds

Extracting initial byte array from file 25, jsSpidersMsd.jpg. This could take several seconds...

Initial JPEG processing time: 3.73 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 25, jsSpidersMsd.jpg

Total detection and processing time for file 25, jsSpidersMsd.jpg: 4.48 seconds

Extracting initial byte array from file 26, jsSpidersSec.jpg. This could take several seconds...

Initial JPEG processing time: 3.69 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 26, jsSpidersSec.jpg

Total detection and processing time for file 26, jsSpidersSec.jpg: 4.47 seconds

Extracting initial byte array from file 27, jsYellowMsd.jpg. This could take several seconds...

Initial JPEG processing time: 2.11 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 27, jsYellowMsd.jpg

Total detection and processing time for file 27, jsYellowMsd.jpg: 2.40 seconds

Extracting initial byte array from file 28, jsYellowSec.jpg. This could take several seconds...

Initial JPEG processing time: 2.12 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg detected for file 28, jsYellowSec.jpg

Total detection and processing time for file 28, jsYellowSec.jpg: 2.34 seconds

Extracting initial byte array from file 29, leaves.jpg. This could take several seconds...

Initial JPEG processing time: 2.23 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 29, leaves.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...

Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 29, leaves.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 29, leaves.jpg

None of the target steganography programs detected for file leaves.jpg

Total detection and processing time for file 29, leaves.jpg: 3.72 seconds

Extracting initial byte array from file 30, moon.jpg. This could take several seconds...

Initial JPEG processing time: 7.88 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 30, moon.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 30, moon.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 30, moon.jpg

None of the target steganography programs detected for file moon.jpg

Total detection and processing time for file 30, moon.jpg: 18.38 seconds

Extracting initial byte array from file 31, ogCowMsd.jpg. This could take several seconds...

Initial JPEG processing time: 7.58 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 31, ogCowMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 31, ogCowMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 31, ogCowMsd.jpg

Total detection and processing time for file 31, ogCowMsd.jpg: 15.84 seconds

Extracting initial byte array from file 32, ogCowSec.jpg. This could take several seconds...

Initial JPEG processing time: 7.66 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 32, ogCowSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 32, ogCowSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 32, ogCowSec.jpg

Total detection and processing time for file 32, ogCowSec.jpg: 15.85 seconds

Extracting initial byte array from file 33, ogCraterMsd.jpg. This could take several seconds...

Initial JPEG processing time: 5.30 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 33, ogCraterMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 33, ogCraterMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 33, ogCraterMsd.jpg

Total detection and processing time for file 33, ogCraterMsd.jpg: 9.04 seconds

Extracting initial byte array from file 34, ogCraterSec.jpg. This could take several seconds...

Initial JPEG processing time: 5.61 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 34, ogCraterSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...

Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 34, ogCraterSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 34, ogCraterSec.jpg

Total detection and processing time for file 34, ogCraterSec.jpg: 9.24 seconds

Extracting initial byte array from file 35, ogForestMsd.jpg. This could take several seconds...

Initial JPEG processing time: 3.88 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 35, ogForestMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...

Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 35, ogForestMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 35, ogForestMsd.jpg

Total detection and processing time for file 35, ogForestMsd.jpg: 5.93 seconds

Extracting initial byte array from file 36, ogForestSec.jpg. This could take several seconds...

Initial JPEG processing time: 3.86 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 36, ogForestSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...

Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 36, ogForestSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 36, ogForestSec.jpg

Total detection and processing time for file 36, ogForestSec.jpg: 5.90 seconds

Extracting initial byte array from file 37, ogGreentreesMsd.jpg. This could take several seconds...

Initial JPEG processing time: 23.65 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 37, ogGreentreesMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...

Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 37, ogGreentreesMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 37, ogGreentreesMsd.jpg

Total detection and processing time for file 37, ogGreentreesMsd.jpg: 46.76 seconds

Extracting initial byte array from file 38, ogGreentreesSec.jpg. This could take several seconds...

Initial JPEG processing time: 23.76 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 38, ogGreentreesSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...

Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 38, ogGreentreesSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 38, ogGreentreesSec.jpg

Total detection and processing time for file 38, ogGreentreesSec.jpg: 46.33 seconds

Extracting initial byte array from file 39, ogHouseplantMsd.jpg. This could take several seconds...

Initial JPEG processing time: 2.25 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 39, ogHouseplantMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...

Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 39, ogHouseplantMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 39, ogHouseplantMsd.jpg

Total detection and processing time for file 39, ogHouseplantMsd.jpg: 3.65 seconds

Extracting initial byte array from file 40, ogHouseplantSec.jpg. This could take several seconds...

Initial JPEG processing time: 2.11 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 40, ogHouseplantSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...

Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 40, ogHouseplantSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 40, ogHouseplantSec.jpg

Total detection and processing time for file 40, ogHouseplantSec.jpg: 3.46 seconds

Extracting initial byte array from file 41, ogPalmsMsd.jpg. This could take several seconds...

Initial JPEG processing time: 1.90 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 41, ogPalmsMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 41, ogPalmsMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 41, ogPalmsMsd.jpg

Total detection and processing time for file 41, ogPalmsMsd.jpg: 3.33 seconds

Extracting initial byte array from file 42, ogPalmsSec.jpg. This could take several seconds...

Initial JPEG processing time: 1.78 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 42, ogPalmsSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 42, ogPalmsSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 42, ogPalmsSec.jpg

Total detection and processing time for file 42, ogPalmsSec.jpg: 3.32 seconds

Extracting initial byte array from file 43, ogPenguinMsd.jpg. This could take several seconds...

Initial JPEG processing time: 5.08 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 43, ogPenguinMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 43, ogPenguinMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 43, ogPenguinMsd.jpg

Total detection and processing time for file 43, ogPenguinMsd.jpg: 10.01 seconds

Extracting initial byte array from file 44, ogPenguinSec.jpg. This could take several seconds...

Initial JPEG processing time: 5.28 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 44, ogPenguinSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 44, ogPenguinSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 44, ogPenguinSec.jpg

Total detection and processing time for file 44, ogPenguinSec.jpg: 10.40 seconds

Extracting initial byte array from file 45, ogPumpkinsMsd.jpg. This could take several seconds...

Initial JPEG processing time: 7.77 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 45, ogPumpkinsMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 45, ogPumpkinsMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 45, ogPumpkinsMsd.jpg

Total detection and processing time for file 45, ogPumpkinsMsd.jpg: 13.90 seconds

Extracting initial byte array from file 46, ogPumpkinsSec.jpg. This could take several seconds...

Initial JPEG processing time: 7.90 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 46, ogPumpkinsSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...

Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 46, ogPumpkinsSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 46, ogPumpkinsSec.jpg

Total detection and processing time for file 46, ogPumpkinsSec.jpg: 13.95 seconds

Extracting initial byte array from file 47, ogSpidersMsd.jpg. This could take several seconds...

Initial JPEG processing time: 3.81 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 47, ogSpidersMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...

Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 47, ogSpidersMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 47, ogSpidersMsd.jpg

Total detection and processing time for file 47, ogSpidersMsd.jpg: 7.67 seconds

Extracting initial byte array from file 48, ogSpidersSec.jpg. This could take several seconds...

Initial JPEG processing time: 3.64 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 48, ogSpidersSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...

Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 48, ogSpidersSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 48, ogSpidersSec.jpg

Total detection and processing time for file 48, ogSpidersSec.jpg: 7.55 seconds

Extracting initial byte array from file 49, ogYellowMsd.jpg. This could take several seconds...

Initial JPEG processing time: 2.12 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 49, ogYellowMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...

Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 49, ogYellowMsd.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 49, ogYellowMsd.jpg

Total detection and processing time for file 49, ogYellowMsd.jpg: 3.35 seconds

Extracting initial byte array from file 50, ogYellowSec.jpg. This could take several seconds...

Initial JPEG processing time: 2.11 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 50, ogYellowSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...

Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 50, ogYellowSec.jpg

*****Final pass - OutGuess: attempting to find signature...

OutGuess (with default options) possible for file 50, ogYellowSec.jpg

Total detection and processing time for file 50, ogYellowSec.jpg: 3.24 seconds

Extracting initial byte array from file 51, palms.jpg. This could take several seconds...

Initial JPEG processing time: 3.22 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 51, palms.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...

Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 51, palms.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 51, palms.jpg

None of the target steganography programs detected for file palms.jpg

Total detection and processing time for file 51, palms.jpg: 4.77 seconds

Extracting initial byte array from file 52, penguin.jpg. This could take several seconds...

Initial JPEG processing time: 9.44 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 52, penguin.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 52, penguin.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 52, penguin.jpg

None of the target steganography programs detected for file penguin.jpg

Total detection and processing time for file 52, penguin.jpg: 14.67 seconds

Extracting initial byte array from file 53, pumpkins.jpg. This could take several seconds...

Initial JPEG processing time: 13.58 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 53, pumpkins.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 53, pumpkins.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 53, pumpkins.jpg

None of the target steganography programs detected for file pumpkins.jpg

Total detection and processing time for file 53, pumpkins.jpg: 20.19 seconds

Extracting initial byte array from file 54, snowhill.jpg. This could take several seconds...

Initial JPEG processing time: 25.85 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 54, snowhill.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 54, snowhill.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 54, snowhill.jpg

None of the target steganography programs detected for file snowhill.jpg

Total detection and processing time for file 54, snowhill.jpg: 36.96 seconds

Extracting initial byte array from file 55, spCowMsd.jpg. This could take several seconds...

Initial JPEG processing time: 10.53 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 55, spCowMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Positive match!

SteganPeg detected for file 55, spCowMsd.jpg

Total detection and processing time for file 55, spCowMsd.jpg: 14.04 seconds

Extracting initial byte array from file 56, spCowSec.jpg. This could take several seconds...

Initial JPEG processing time: 10.67 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 56, spCowSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Positive match!

SteganPeg detected for file 56, spCowSec.jpg

Total detection and processing time for file 56, spCowSec.jpg: 14.41 seconds

Extracting initial byte array from file 57, spCraterMsd.jpg. This could take several seconds...

Initial JPEG processing time: 8.76 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 57, spCraterMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Positive match!

SteganPeg detected for file 57, spCraterMsd.jpg

Total detection and processing time for file 57, spCraterMsd.jpg: 10.62 seconds

Extracting initial byte array from file 58, spCraterSec.jpg. This could take several seconds...

Initial JPEG processing time: 8.75 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 58, spCraterSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Negative match...
Checking password 'qwerty'... Negative match...
Checking password 'abc123'... Negative match...
Checking password 'Million2'... Positive match!

SteganPeg detected for file 58, spCraterSec.jpg

Total detection and processing time for file 58, spCraterSec.jpg: 11.10 seconds

Extracting initial byte array from file 59, spForestMsd.jpg. This could take several seconds...

Initial JPEG processing time: 5.89 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 59, spForestMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Positive match!

SteganPeg detected for file 59, spForestMsd.jpg

Total detection and processing time for file 59, spForestMsd.jpg: 6.96 seconds

Extracting initial byte array from file 60, spForestSec.jpg. This could take several seconds...

Initial JPEG processing time: 5.84 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 60, spForestSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Negative match...
Checking password 'qwerty'... Negative match...

Checking password 'abc123'... Negative match...
Checking password 'Million2'... Negative match...
Checking password '000000'... Negative match...
Checking password '1234'... Negative match...
Checking password 'iloveyou'... Negative match...
Checking password 'aaron431'... Negative match...
Checking password 'password1'... Positive match!

SteganPeg detected for file 60, spForestSec.jpg

Total detection and processing time for file 60, spForestSec.jpg: 7.43 seconds

Extracting initial byte array from file 61, spGreentreesMsd.jpg. This could take several seconds...

Initial JPEG processing time: 39.13 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 61, spGreentreesMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Negative match...
Checking password 'qwerty'... Negative match...
Checking password 'abc123'... Negative match...
Checking password 'Million2'... Negative match...
Checking password '000000'... Positive match!

SteganPeg detected for file 61, spGreentreesMsd.jpg

Total detection and processing time for file 61, spGreentreesMsd.jpg: 48.92 seconds

Extracting initial byte array from file 62, spGreentreesSec.jpg. This could take several seconds...

Initial JPEG processing time: 39.92 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 62, spGreentreesSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Positive match!

SteganPeg detected for file 62, spGreentreesSec.jpg

Total detection and processing time for file 62, spGreentreesSec.jpg: 49.22 seconds

Extracting initial byte array from file 63, spHouseplantMsd.jpg. This could take several seconds...

Initial JPEG processing time: 2.46 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 63, spHouseplantMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...

Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Negative match...
Checking password 'qwerty'... Negative match...
Checking password 'abc123'... Positive match!

SteganPeg detected for file 63, spHouseplantMsd.jpg

Total detection and processing time for file 63, spHouseplantMsd.jpg: 3.43 seconds

Extracting initial byte array from file 64, spHouseplantSec.jpg. This could take several seconds...

Initial JPEG processing time: 2.37 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 64, spHouseplantSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Negative match...

Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Negative match...
Checking password 'aaron431'... Negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Positive match!

SteganPeg detected for file 64, spHouseplantSec.jpg

Total detection and processing time for file 64, spHouseplantSec.jpg: 3.07 seconds

Extracting initial byte array from file 65, spiders.jpg. This could take several seconds...

Initial JPEG processing time: 5.44 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 65, spiders.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...

Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 65, spiders.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 65, spiders.jpg

None of the target steganography programs detected for file spiders.jpg

Total detection and processing time for file 65, spiders.jpg: 9.44 seconds

Extracting initial byte array from file 66, spPalmsMsd.jpg. This could take several seconds...

Initial JPEG processing time: 3.11 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 66, spPalmsMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Negative match...

Checking password 'qwerty'... Positive match!

SteganPeg detected for file 66, spPalmsMsd.jpg

Total detection and processing time for file 66, spPalmsMsd.jpg: 4.03 seconds

Extracting initial byte array from file 67, spPalmsSec.jpg. This could take several seconds...

Initial JPEG processing time: 3.07 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 67, spPalmsSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Positive match!

SteganPeg detected for file 67, spPalmsSec.jpg

Total detection and processing time for file 67, spPalmsSec.jpg: 3.95 seconds

Extracting initial byte array from file 68, spPenguinMsd.jpg. This could take several seconds...

Initial JPEG processing time: 9.44 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 68, spPenguinMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...

Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Positive match!

SteganPeg detected for file 68, spPenguinMsd.jpg

Total detection and processing time for file 68, spPenguinMsd.jpg: 11.70 seconds

Extracting initial byte array from file 69, spPenguinSec.jpg. This could take several seconds...

Initial JPEG processing time: 9.47 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 69, spPenguinSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Positive match!

SteganPeg detected for file 69, spPenguinSec.jpg

Total detection and processing time for file 69, spPenguinSec.jpg: 11.75 seconds

Extracting initial byte array from file 70, spPumpkinsMsd.jpg. This could take several seconds...

Initial JPEG processing time: 13.38 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 70, spPumpkinsMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...

Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Negative match...
Checking password 'qwerty'... Negative match...
Checking password 'abc123'... Negative match...
Checking password 'Million2'... Negative match...
Checking password '000000'... Negative match...
Checking password '1234'... Negative match...
Checking password 'iloveyou'... Positive match!

SteganPeg detected for file 70, spPumpkinsMsd.jpg

Total detection and processing time for file 70, spPumpkinsMsd.jpg: 16.87 seconds

Extracting initial byte array from file 71, spPumpkinsSec.jpg. This could take several seconds...

Initial JPEG processing time: 13.80 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 71, spPumpkinsSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...

Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Negative match...
Checking password 'qwerty'... Negative match...
Checking password 'abc123'... Negative match...
Checking password 'Million2'... Negative match...
Checking password '000000'... Negative match...
Checking password '1234'... Positive match!

SteganPeg detected for file 71, spPumpkinsSec.jpg

Total detection and processing time for file 71, spPumpkinsSec.jpg: 17.06 seconds

Extracting initial byte array from file 72, spSpidersMsd.jpg. This could take several seconds...

Initial JPEG processing time: 5.71 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 72, spSpidersMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Positive match!

SteganPeg detected for file 72, spSpidersMsd.jpg

Total detection and processing time for file 72, spSpidersMsd.jpg: 7.28 seconds

Extracting initial byte array from file 73, spSpidersSec.jpg. This could take several seconds...

Initial JPEG processing time: 5.53 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 73, spSpidersSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Negative match...
Checking password '1234567'... Negative match...
Checking password 'qwerty'... Negative match...
Checking password 'abc123'... Negative match...
Checking password 'Million2'... Negative match...
Checking password '000000'... Negative match...
Checking password '1234'... Negative match...
Checking password 'iloveyou'... Negative match...
Checking password 'aaron431'... Positive match!

SteganPeg detected for file 73, spSpidersSec.jpg

Total detection and processing time for file 73, spSpidersSec.jpg: 7.93 seconds

Extracting initial byte array from file 74, spYellowMsd.jpg. This could take several seconds...

Initial JPEG processing time: 3.17 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 74, spYellowMsd.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...

Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Positive match!

SteganPeg detected for file 74, spYellowMsd.jpg

Total detection and processing time for file 74, spYellowMsd.jpg: 3.86 seconds

Extracting initial byte array from file 75, spYellowSec.jpg. This could take several seconds...

Initial JPEG processing time: 3.22 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 75, spYellowSec.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Negative match...
Checking password '123456789'... Negative match...
Checking password 'picture1'... Negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Negative match...
Checking password '111111'... Negative match...
Checking password '123123'... Negative match...
Checking password '12345'... Negative match...
Checking password '1234567890'... Negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Positive match!

SteganPeg detected for file 75, spYellowSec.jpg

Total detection and processing time for file 75, spYellowSec.jpg: 3.97 seconds

Extracting initial byte array from file 76, squirrel.jpg. This could take several seconds...

Initial JPEG processing time: 18.09 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 76, squirrel.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 76, squirrel.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 76, squirrel.jpg

None of the target steganography programs detected for file squirrel.jpg

Total detection and processing time for file 76, squirrel.jpg: 31.12 seconds

Extracting initial byte array from file 77, sunset.jpg. This could take several seconds...

Initial JPEG processing time: 1.96 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 77, sunset.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...

Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 77, sunset.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 77, sunset.jpg

None of the target steganography programs detected for file sunset.jpg

Total detection and processing time for file 77, sunset.jpg: 3.59 seconds

Extracting initial byte array from file 78, tiger.jpg. This could take several seconds...

Initial JPEG processing time: 4.69 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 78, tiger.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...

Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 78, tiger.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 78, tiger.jpg

None of the target steganography programs detected for file tiger.jpg

Total detection and processing time for file 78, tiger.jpg: 6.55 seconds

Extracting initial byte array from file 79, whitehouse.jpg. This could take several seconds...

Initial JPEG processing time: 10.99 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 79, whitehouse.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...
Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...

Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 79, whitehouse.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 79, whitehouse.jpg

None of the target steganography programs detected for file whitehouse.jpg

Total detection and processing time for file 79, whitehouse.jpg: 15.62 seconds

Extracting initial byte array from file 80, yellow.jpg. This could take several seconds...

Initial JPEG processing time: 3.27 seconds

*****First pass - Jsteg: attempting to find signature...

Jsteg not detected for file 80, yellow.jpg

*****Second pass - SteganPeg: attempting to find signature using common passwords...
Checking password '123456'... Data length too big, negative match...
Checking password '123456789'... Data length too big, negative match...
Checking password 'picture1'... Data length too big, negative match...
Checking password 'password'... Data length too big, negative match...

Checking password '12345678'... Data length too big, negative match...
Checking password '111111'... Data length too big, negative match...
Checking password '123123'... Data length too big, negative match...
Checking password '12345'... Data length too big, negative match...
Checking password '1234567890'... Data length too big, negative match...
Checking password 'senha'... Data length too big, negative match...
Checking password '1234567'... Data length too big, negative match...
Checking password 'qwerty'... Data length too big, negative match...
Checking password 'abc123'... Data length too big, negative match...
Checking password 'Million2'... Data length too big, negative match...
Checking password '000000'... Data length too big, negative match...
Checking password '1234'... Data length too big, negative match...
Checking password 'iloveyou'... Data length too big, negative match...
Checking password 'aaron431'... Data length too big, negative match...
Checking password 'password1'... Data length too big, negative match...
Checking password 'qqww1122'... Data length too big, negative match...

SteganPeg not detected for file 80, yellow.jpg

*****Final pass - OutGuess: attempting to find signature...
Seed too large...

OutGuess (with default options) not possible for file 80, yellow.jpg

None of the target steganography programs detected for file yellow.jpg

Total detection and processing time for file 80, yellow.jpg: 4.54 seconds

Analysis is complete. Log file is jrpegLog0.txt, and metrics are saved in jrpegStats0.csv

APPENDIX B

LOG FILE 2:CSV FILE

File	Size	Height	Width	Jsteg	SteganPEG	OutGuess	CoeffTime	OverallTime
cactus.jpg	591155	3600	2400	FALSE	FALSE	FALSE	4.97	11.23
cow.jpg	1275990	2667	4000	FALSE	FALSE	FALSE	10.33	18.54
crater.jpg	1072272	1703	2556	FALSE	FALSE	FALSE	8.52	12.4
eagle.jpg	770867	1634	2400	FALSE	FALSE	FALSE	5.97	9.23
forest.jpg	729925	1280	1920	FALSE	FALSE	FALSE	5.54	7.71
frog.jpg	298370	1805	2400	FALSE	FALSE	FALSE	2.49	5.56
greentrees.jpg	4664976	4293	6440	FALSE	FALSE	FALSE	38.45	62.32
houseplant.jpg	316713	1066	1599	FALSE	FALSE	FALSE	2.42	3.79
jsCowMsd.jpg	822003	2667	4000	TRUE	FALSE	FALSE	7.01	8.61
jsCowSec.jpg	822010	2667	4000	TRUE	FALSE	FALSE	7.41	8.96
jsCraterMsd.jpg	601331	1703	2556	TRUE	FALSE	FALSE	5.11	5.79
jsCraterSec.jpg	601336	1703	2556	TRUE	FALSE	FALSE	5.31	6.04
jsForestMsd.jpg	458084	1280	1920	TRUE	FALSE	FALSE	3.78	4.18
jsForestSec.jpg	458099	1280	1920	TRUE	FALSE	FALSE	3.76	4.16
jsGreentreesMsd.jpg	2650002	4293	6440	TRUE	FALSE	FALSE	23.06	27.24
jsGreentreesSec.jpg	2650020	4293	6440	TRUE	FALSE	FALSE	23.52	27.83
jsHouseplantMsd.jpg	227946	1066	1599	TRUE	FALSE	FALSE	1.95	2.22
jsHouseplantSec.jpg	255675	1066	1599	TRUE	FALSE	FALSE	2.2	2.46
jsPalmsMsd.jpg	216005	1200	1600	TRUE	FALSE	FALSE	1.79	2.08
jsPalmsSec.jpg	216003	1200	1600	TRUE	FALSE	FALSE	1.8	2.09
jsPenguinMsd.jpg	591038	3037	2025	TRUE	FALSE	FALSE	5.03	6.04
jsPenguinSec.jpg	591050	3037	2025	TRUE	FALSE	FALSE	5.18	6.28
jsPumpkinsMsd.jpg	914289	3239	2293	TRUE	FALSE	FALSE	8.01	9.4
jsPumpkinsSec.jpg	914303	3239	2293	TRUE	FALSE	FALSE	7.77	8.93
jsSpidersMsd.jpg	419257	2108	2400	TRUE	FALSE	FALSE	3.73	4.48
jsSpidersSec.jpg	419279	2108	2400	TRUE	FALSE	FALSE	3.69	4.47
jsYellowMsd.jpg	256814	1200	1200	TRUE	FALSE	FALSE	2.11	2.4
jsYellowSec.jpg	256821	1200	1200	TRUE	FALSE	FALSE	2.12	2.34
leaves.jpg	306227	1066	1599	FALSE	FALSE	FALSE	2.23	3.72
moon.jpg	875648	2996	4489	FALSE	FALSE	FALSE	7.88	18.38
ogCowMsd.jpg	823975	2667	4000	FALSE	FALSE	TRUE	7.58	15.84
ogCowSec.jpg	823575	2667	4000	FALSE	FALSE	TRUE	7.66	15.85
ogCraterMsd.jpg	600659	1703	2556	FALSE	FALSE	TRUE	5.3	9.04
ogCraterSec.jpg	600482	1703	2556	FALSE	FALSE	TRUE	5.61	9.24
ogForestMsd.jpg	458562	1280	1920	FALSE	FALSE	TRUE	3.88	5.93
ogForestSec.jpg	458322	1280	1920	FALSE	FALSE	TRUE	3.86	5.9
ogGreentreesMsd.jpg	2648358	4293	6440	FALSE	FALSE	TRUE	23.65	46.76
ogGreentreesSec.jpg	2648133	4293	6440	FALSE	FALSE	TRUE	23.76	46.33
ogHouseplantMsd.jpg	254165	1066	1599	FALSE	FALSE	TRUE	2.25	3.65
ogHouseplantSec.jpg	253888	1066	1599	FALSE	FALSE	TRUE	2.11	3.46

ogPalmsMsd.jpg	216598	1200	1600	FALSE	FALSE	TRUE	1.9	3.33
ogPalmsSec.jpg	216295	1200	1600	FALSE	FALSE	TRUE	1.78	3.32
ogPenguinMsd.jpg	591951	3037	2025	FALSE	FALSE	TRUE	5.08	10.01
ogPenguinSec.jpg	591768	3037	2025	FALSE	FALSE	TRUE	5.28	10.4
ogPumpkinsMsd.jpg	912251	3239	2293	FALSE	FALSE	TRUE	7.77	13.9
ogPumpkinsSec.jpg	912083	3239	2293	FALSE	FALSE	TRUE	7.9	13.95
ogSpidersMsd.jpg	421724	2108	2400	FALSE	FALSE	TRUE	3.81	7.67
ogSpidersSec.jpg	421149	2108	2400	FALSE	FALSE	TRUE	3.64	7.55
ogYellowMsd.jpg	258233	1200	1200	FALSE	FALSE	TRUE	2.12	3.35
ogYellowSec.jpg	257896	1200	1200	FALSE	FALSE	TRUE	2.11	3.24
palms.jpg	380985	1200	1600	FALSE	FALSE	FALSE	3.22	4.77
penguin.jpg	1126912	3037	2025	FALSE	FALSE	FALSE	9.44	14.67
pumpkins.jpg	1652888	3239	2293	FALSE	FALSE	FALSE	13.58	20.19
snowhill.jpg	3131428	2992	3992	FALSE	FALSE	FALSE	25.85	36.96
spCowMsd.jpg	1276041	2667	4000	FALSE	TRUE	FALSE	10.53	14.04
spCowSec.jpg	1276026	2667	4000	FALSE	TRUE	FALSE	10.67	14.41
spCraterMsd.jpg	1072285	1703	2556	FALSE	TRUE	FALSE	8.76	10.62
spCraterSec.jpg	1072279	1703	2556	FALSE	TRUE	FALSE	8.75	11.1
spForestMsd.jpg	729943	1280	1920	FALSE	TRUE	FALSE	5.89	6.96
spForestSec.jpg	729937	1280	1920	FALSE	TRUE	FALSE	5.84	7.43
spGreentreesMsd.jpg	4664980	4293	6440	FALSE	TRUE	FALSE	39.13	48.92
spGreentreesSec.jpg	4664978	4293	6440	FALSE	TRUE	FALSE	39.92	49.22
spHouseplantMsd.jpg	316729	1066	1599	FALSE	TRUE	FALSE	2.46	3.43
spHouseplantSec.jpg	316725	1066	1599	FALSE	TRUE	FALSE	2.37	3.07
spiders.jpg	673936	2108	2400	FALSE	FALSE	FALSE	5.44	9.44
spPalmsMsd.jpg	381002	1200	1600	FALSE	TRUE	FALSE	3.11	4.03
spPalmsSec.jpg	381005	1200	1600	FALSE	TRUE	FALSE	3.07	3.95
spPenguinMsd.jpg	1126920	3037	2025	FALSE	TRUE	FALSE	9.44	11.7
spPenguinSec.jpg	1126916	3037	2025	FALSE	TRUE	FALSE	9.47	11.75
spPumpkinsMsd.jpg	1652906	3239	2293	FALSE	TRUE	FALSE	13.38	16.87
spPumpkinsSec.jpg	1652886	3239	2293	FALSE	TRUE	FALSE	13.8	17.06
spSpidersMsd.jpg	674067	2108	2400	FALSE	TRUE	FALSE	5.71	7.28
spSpidersSec.jpg	674035	2108	2400	FALSE	TRUE	FALSE	5.53	7.93
spYellowMsd.jpg	400656	1200	1200	FALSE	TRUE	FALSE	3.17	3.86
spYellowSec.jpg	400641	1200	1200	FALSE	TRUE	FALSE	3.22	3.97
squirrel.jpg	2180916	3190	4785	FALSE	FALSE	FALSE	18.09	31.12
sunset.jpg	230704	1080	1920	FALSE	FALSE	FALSE	1.96	3.59
tiger.jpg	581934	1312	1474	FALSE	FALSE	FALSE	4.69	6.55
whitehouse.jpg	1361001	1944	2592	FALSE	FALSE	FALSE	10.99	15.62
yellow.jpg	400590	1200	1200	FALSE	FALSE	FALSE	3.27	4.54

APPENDIX C

JREVEALPEG CODE

jrpeg.py

```
#JRevealPeg - a semi-blind JPEG steganalysis tool that targets known open-source embedding programs.
#Main Script: jrpeg.py
#Author: Charles Badami
#Language: Python 3.7.7
#Custom Module Dependencies: coefx.py, jsdec.py, spdec.py, ogdec.py; should be saved in the same folder as jrpeg.py
#Usage: From Windows cmd line, type "python jrpeg.py". Then, choose from the menu options.
#----- Analyzes single files or a folder of multiple files

#################################################################################################################

##Imports

#For measuring processing time
import time

#For directory manipulation
import os


#Custom JPEG processing and byte extraction module
import coefx


#Custom analysis modules for target steganography programs#

#Jsteg
import jsdec

#SteganPeg
import spdec

#OutGuess
import ogdec


#Steganalysis routine
#----- Best detection order is Jsteg, SteganPeg, then likelihood of OutGuess (there is no definite signature, just heuristics)
def analyze(srcFiles, log):

    log.lprint("Number of files to be analyzed: " + str(len(srcFiles)))

    count = 0
    for f in srcFiles:
        count += 1

        #Extract array of Huffman-decoded scan data from input image (plus height, width -> y, x)
        log.lprint("\nExtracting initial byte array from file " + str(count) + ", " + f + ". This could take several seconds...")

        startTime = time.perf_counter()

        try:
            coeffs, y, x = coefx.extract(f)
            if (coeffs == None):
                log.lprint("\tNo scan data! May be bad file type.")
                continue
        except FileNotFoundError:
            log.lprint("Error: File not found! : " + f)
            continue
        except Exception as inst:
            log.lprint("Error: File might not be an accepted type of JPEG (baseline, 4:2:0 (2x2) subsampling) : " + f)
            m,n = inst.args
            log.lprint("\t" + m + " " + n)
            continue

        jpgProcTime = time.perf_counter() - startTime
        log.lprint("\nInitial JPEG processing time: {:.2f} seconds".format(jpgProcTime))

        #Initialize detection results to false
        js = False
        sp = False
        og = False
        message = "\nNone of the target steganography programs detected for file " + f
```

```
        #Attempt to identify Jsteg signature first, if not then SteganPeg, if not then OutGuess
        log.lprint("\n*****First pass - Jsteg: attempting to find signature...")
        try:
            js = jsdec.detect(coeffs)
        except Exception as inst:
            m,n = inst.args
            log.lprint("\t" + m + " " + n)
            log.lprint("\nThere was a problem with Jsteg detection. Moving to next target...")
        if (js):
            message = "\nJsteg detected for file " + str(count) + ", " + f
        else:
            log.lprint("\nJsteg not detected for file " + str(count) + ", " + f)
            log.lprint("\n*****Second pass - SteganPeg: attempting to find signature using common passwords...")
            try:
                sp, msgs = spdec.detect(coeffs)
                log.lprint(msgs)
            except Exception as inst:
                m,n = inst.args
                log.lprint("\t" + m + " " + n)
                log.lprint("\nThere was a problem with SteganPeg detection. Moving to next target...")
            if (sp):
                message = "\nSteganPeg detected for file " + str(count) + ", " + f
            else:
                log.lprint("\nSteganPeg not detected for file " + str(count) + ", " + f)
                log.lprint("\n*****Final pass - OutGuess: attempting to find signature...")
                ogErr = False
                try:
                    og, msgs = ogdec.detect(coeffs, y, x)
                    log.lprint(msgs)
                except Exception as inst:
                    m,n = inst.args
                    log.lprint("\t" + m + " " + n)
                    log.lprint("\nThere was a problem with OutGuess detection. Finalizing...")
                    ogErr = True
                if (og):
                    message = "\nOutGuess (with default options) possible for file " + str(count) + ", " + f
                elif (not ogErr):
                    log.lprint("\nOutGuess (with default options) not possible for file " + str(count) + ", " + f)

        endTime = time.perf_counter()
        totalTime = (endTime -startTime)
        log.lprint(message)
        log.lprint("\nTotal detection and processing time for file "+ str(count) + ", " + f + ": {:.2f} seconds".format(totalTime))

        #Write to csv file
        fSize = os.path.getsize(f)
        log.csvFile.write(f + "," + str(fSize) + "," + str(y) + "," + str(x) + "," + str(js) + "," + str(sp) + "," + str(og) +
                          ",{:.2f},{:.2f}\n".format(jpgProcTime, totalTime))

    log.lprint("\nAnalysis is complete. Log file is " + log.logName + ", and metrics are saved in " + log.csvName)
    log.logFile.close()
    log.csvFile.close()


#Logging class
class Logger:

    def __init__(self, cwd):
        self.cwd = cwd
        self.setLogFile()
        self.setCSVFile()

    def setLogFile(self):
        path = os.path.join(self.cwd, "logs")
        (_, _, files) = next(os.walk(path))
        #jrpegLogX.txt
        for c in range(len(files)):
            fname = "jrpegLog"+str(c)+".txt"
            if (fname not in files):
                self.logFile = open(path+'/'+fname, "w")
                self.logFile.write("Logged on " + time.asctime( time.localtime(time.time()) )+"\n")
                self.logName = fname
                break
            c += 1

    def setCSVFile(self):
        path = os.path.join(self.cwd, "logs")
        (_, _, files) = next(os.walk(path))
        #jrpegStatsX.csv
        for c in range(len(files)):
            fname = "jrpegStats"+str(c)+".csv"
            if (fname not in files):
                self.csvFile = open(path+'/'+fname, "w")
                self.csvFile.write("File,Size,Height,Width,Jsteg,SteganPEG,OutGuess,CoeffTime,OverallTime"+"\n")
                self.csvName = fname
                break
            c += 1

    def lprint(self, line):
        print(line)
        self.logFile.write(line + "\n")
```

```
#User menu prompt for main program
def menu():
    print("\n(f) Enter a path to a single file for analysis")
    print("(d) Enter a path to a folder to analyze multiple files")
    print("(h) Help")
    print("(q) Quit")
    return input("\nEnter a menu option: ")


##### Main program #####

#Save current working directory
cwd = os.getcwd()

#Initialize /logs directory if necessary
(_, _, files) = next(os.walk(cwd))
try:
    path = os.path.join(cwd, "logs")
    os.mkdir(path)
except:
    pass

print("\n**************************************************************************\n")
print("   Welcome to JRevealPEG, a semi-blind steganalysis tool for JPEGs\n")
print("**************************************************************************\n")
choice = menu()

while (choice.lower() != 'q'):


    if (choice not in ['f','d','h','q']):
        print("Please enter a valid choice...\n")
        choice = menu()
        continue

    if (choice == 'f'):
        path = input("Enter a file path: ")
        if (os.path.isfile(path)):
            log = Logger(cwd)
            log.lprint("File path entered: " + path)
            analyze([path], log)
        else:
            print("Not a valid file path.")
            choice = menu()
            continue

    if (choice == 'd'):
        path = input("Enter a directory for input files: ")
        if (os.path.isdir(path)):
            (_, _, srcFiles) = next(os.walk(path))
            os.chdir(path)
            #Initialize logger
            log = Logger(cwd)
            log.lprint("Directory entered: " + path)
            analyze(srcFiles, log)
        else:
            print("Not a valid directory.")
            choice = menu()
            continue

    if (choice == 'h'):
        print("\nJRevealPEG will attempt to detect steganography in JPEGs given as input.")
        print("This program does NOT extract the hidden data.")
        print("Currently, this program is only be able to detect signatures from Jsteg 0.3.0, SteganPeg 1.0, and OutGuess 0.2.2.")
        print("Current JPEG compatibility: Baseline DCT, with 4:2:0 (2x2) chroma subsampling only.")
        print("\nYou can supply the path to a single file, or to a folder with multiple files.")
        print("The analysis report is saved in a log file, and detection stats are also saved in a csv file.")
        print("\nDetection details:")
        print("1. Detection will occur in this order: Jsteg, SteganPEG, OutGuess.")
        print("---Once a signature is detected, analysis is complete for that file.")
        print("2. Assumptions/Preconditions:")
        print("---Jsteg: No password used with this tool.")
        print("---SteganPEG: Assumes password is from top 20 list of 2020 by NordPass.")
        print("---OutGuess: Assumes default options (key is \"Default key\"). Result is either \"possible\" or \"not possible\".")
        print("3. For all input files, this program assumes there is only one hidden message or file.")


    choice = menu()
```

coefx.py

```python
#JPEG quantized coefficient extractor, custom module for JRevealPEG (jrpeg.py)
#Author: Charles Badami
#Credit: Much of this code is copied, adapted and revised from https://github.com/yasoob/Baseline-JPEG-Decoder by M.Yasoob Ullah Khalid
#Current compatibility: only JPEGs with baseline encoding and 4:2:0 chroma subsampling (YYYYCbCr each MCU)


from struct import unpack
import math


def RemoveFF00(data):
    """
    Removes 0x00 after 0xff in the image scan section of JPEG
    """
    datapro = []
    i = 0
    while True:
        b, bnext = unpack("BB", data[i : i + 2])
        if b == 0xFF:
            if bnext != 0:
                break
            datapro.append(data[i])
            i += 2
        else:
            datapro.append(data[i])
            i += 1
    return datapro, i


def GetArray(type, l, length):
    """
    A convenience function for unpacking an array from bitstream
    """
    s = ""
    for i in range(length):
        s = s + type
    return list(unpack(s, l[:length]))


def DecodeNumber(code, bits):
    l = 2 ** (code - 1)
    if bits >= l:
        return bits
    else:
        return bits - (2 * l - 1)


class HuffmanTable:
    """
    A Huffman Table class
    """

    def __init__(self):
        self.root = []
        self.elements = []

    def BitsFromLengths(self, root, element, pos):
        if isinstance(root, list):
            if pos == 0:
                if len(root) < 2:
                    root.append(element)
                    return True
                return False
            for i in [0, 1]:
                if len(root) == i:
                    root.append([])
                if self.BitsFromLengths(root[i], element, pos - 1) == True:
                    return True
        return False

    def GetHuffmanBits(self, lengths, elements):
        self.elements = elements
        ii = 0
        for i in range(len(lengths)):
            for j in range(lengths[i]):
                self.BitsFromLengths(self.root, elements[ii], i)
                ii += 1

    def Find(self, st):
        r = self.root
        while isinstance(r, list):
            r = r[st.GetBit()]
        return r

    def GetCode(self, st):
        while True:
            res = self.Find(st)
            if res == 0:
                return 0
            elif res != -1:
                return res
```

```
class Stream:
    """
    A bit stream class with convenience methods
    """

    def __init__(self, data):
        self.data = data
        self.pos = 0

    def GetBit(self):
        b = self.data[self.pos >> 3]
        s = 7 - (self.pos & 0x7)
        self.pos += 1
        return (b >> s) & 1

    def GetBitN(self, l):
        val = 0
        for _ in range(l):
            val = val * 2 + self.GetBit()
        return val

    def len(self):
        return len(self.data)


class JPEG:
    """
    JPEG class for decoding a baseline encoded JPEG image (to pre-Huffman encoding stage only)
    """

    def __init__(self, image_file):
        self.huffman_tables = {}
        self.quant = {}
        self.quantMapping = []
        with open(image_file, "rb") as f:
            self.img_data = f.read()

    def DefineQuantizationTables(self, data):
        (hdr,) = unpack("B", data[0:1])
        self.quant[hdr] = GetArray("B", data[1 : 1 + 64], 64)
        data = data[65:]

    def BuildMatrix(self, st, idx, quant, olddccoeff):
        #Current block of MCU coefficients
        cBlock = [0] * 64

        code = self.huffman_tables[0 + idx].GetCode(st)
        bits = st.GetBitN(code)
        dccoeff = DecodeNumber(code, bits) + olddccoeff

        cBlock[0] = dccoeff

        l = 1
        while l < 64:
            code = self.huffman_tables[16 + idx].GetCode(st)
            if code == 0:
                break

            # The first part of the AC key_len
            # is the number of leading zeros
            if code > 15:
                l += code >> 4
                code = code & 0x0F

            bits = st.GetBitN(code)

            if l < 64:
                coeff = DecodeNumber(code, bits)

                cBlock[l] = coeff

                l += 1

        return cBlock, dccoeff


    def BaselineDCT(self, data):
        hdr, self.height, self.width, components = unpack(">BHHB", data[0:6])

        #In case width and/or height not divisible by 16
        self.extWidth = self.width
        self.extHeight = self.height
        if (self.width%16 != 0):
            self.extWidth += (16-(self.width%16))
        if (self.height%16 != 0):
            self.extHeight += (16-(self.height%16))

        for i in range(components):
            id, samp, QtbId = unpack("BBB", data[6 + i * 3 : 9 + i * 3])
            self.quantMapping.append(QtbId)
```

```
    def decodeHuffman(self, data):
        offset = 0
        (header,) = unpack("B", data[offset : offset + 1])
        offset += 1

        lengths = GetArray("B", data[offset : offset + 16], 16)
        offset += 16

        elements = []
        for i in lengths:
            elements += GetArray("B", data[offset : offset + i], i)
            offset += i

        hf = HuffmanTable()
        hf.GetHuffmanBits(lengths, elements)
        self.huffman_tables[header] = hf

        data = data[offset:]

    def checkHtStructType(self, len_chunk, data):
        tableLengths = data[5:21]
        valSum = 0
        for val in tableLengths:
            valSum += val
        if ((len_chunk - 0x13) == valSum):
            len_chunk += 2
            chunk = data[4:len_chunk]
            self.decodeHuffman(chunk)
            return len_chunk, chunk, data
        else:
            #Only first DHT has the marker (0xFFC4)
            tableLengths = data[5:21]
            len_chunk = valSum + 0x13 + 2
            chunk = data[4:len_chunk]
            self.decodeHuffman(chunk)
            data = data[len_chunk:]
            for i in range(3):
                tableLengths = data[1:17]
                valSum = 0
                for val in tableLengths:
                    valSum += val
                len_chunk = valSum + 0x13 - 2 #header adjustment
                chunk = data[:len_chunk] #start at index 0 cause no header here
                self.decodeHuffman(chunk)
                data = data[len_chunk:]
            return len_chunk, chunk, data

    #Main image scan data
    def StartOfScan(self, data, hdrlen):
        data, lenchunk = RemoveFF00(data[hdrlen:])

        #Pad scan data with extra 0 byte in case SteganPeg is short
        data.append(0)

        st = Stream(data)
        oldlumdccoeff, oldCbdccoeff, oldCrdccoeff = 0, 0, 0

        #List of decoded MCU matrices
        mcuList = []

        #Configured for 4:2:0 chroma subsampling only
        for y in range(self.extHeight // 16):
            for x in range(self.extWidth // 16):

                #First decode 4 Y components in a row
                for i in range(4):
                    cBlock, oldlumdccoeff = self.BuildMatrix(
                        st, 0, self.quant[self.quantMapping[0]], oldlumdccoeff
                    )
                    mcuList.append(cBlock)

                #Now decode Cb, Cr components
                cBlock, oldCbdccoeff = self.BuildMatrix(
                    st, 1, self.quant[self.quantMapping[1]], oldCbdccoeff
                )
                mcuList.append(cBlock)

                cBlock, oldCrdccoeff = self.BuildMatrix(
                    st, 1, self.quant[self.quantMapping[2]], oldCrdccoeff
                )
                mcuList.append(cBlock)

        return mcuList


    def decode(self):
        data = self.img_data

        #Check for Start of Image marker at beginning of file
        (marker,) = unpack(">H", data[0:2])
        if (marker != 0xFFD8):
            raise Exception("No SOI marker")
```

```
    def decodeHuffman(self, data):
        offset = 0
        (header,) = unpack("B", data[offset : offset + 1])
        offset += 1

        lengths = GetArray("B", data[offset : offset + 16], 16)
        offset += 16

        elements = []
        for i in lengths:
            elements += GetArray("B", data[offset : offset + i], i)
            offset += i

        hf = HuffmanTable()
        hf.GetHuffmanBits(lengths, elements)
        self.huffman_tables[header] = hf

        data = data[offset:]

    def checkHtStructType(self, len_chunk, data):
        tableLengths = data[5:21]
        valSum = 0
        for val in tableLengths:
            valSum += val
        if ((len_chunk - 0x13) == valSum):
            len_chunk += 2
            chunk = data[4:len_chunk]
            self.decodeHuffman(chunk)
            return len_chunk, chunk, data
        else:
            #Only first DHT has the marker (0xFFC4)
            tableLengths = data[5:21]
            len_chunk = valSum + 0x13 + 2
            chunk = data[4:len_chunk]
            self.decodeHuffman(chunk)
            data = data[len_chunk:]
            for i in range(3):
                tableLengths = data[1:17]
                valSum = 0
                for val in tableLengths:
                    valSum += val
                len_chunk = valSum + 0x13 - 2 #header adjustment
                chunk = data[:len_chunk] #start at index 0 cause no header here
                self.decodeHuffman(chunk)
                data = data[len_chunk:]
            return len_chunk, chunk, data

    #Main image scan data
    def StartOfScan(self, data, hdrlen):
        data, lenchunk = RemoveFF00(data[hdrlen:])

        #Pad scan data with extra 0 byte in case SteganPeg is short
        data.append(0)

        st = Stream(data)
        oldlumdccoeff, oldCbdccoeff, oldCrdccoeff = 0, 0, 0

        #List of decoded MCU matrices
        mcuList = []

        #Configured for 4:2:0 chroma subsampling only
        for y in range(self.extHeight // 16):
            for x in range(self.extWidth // 16):

                #First decode 4 Y components in a row
                for i in range(4):
                    cBlock, oldlumdccoeff = self.BuildMatrix(
                        st, 0, self.quant[self.quantMapping[0]], oldlumdccoeff
                    )
                    mcuList.append(cBlock)

                #Now decode Cb, Cr components
                cBlock, oldCbdccoeff = self.BuildMatrix(
                    st, 1, self.quant[self.quantMapping[1]], oldCbdccoeff
                )
                mcuList.append(cBlock)

                cBlock, oldCrdccoeff = self.BuildMatrix(
                    st, 1, self.quant[self.quantMapping[2]], oldCrdccoeff
                )
                mcuList.append(cBlock)

        return mcuList
```

```
    def decode(self):
        data = self.img_data

        #Check for Start of Image marker at beginning of file
        (marker,) = unpack(">H", data[0:2])
        if (marker != 0xFFD8):
            raise Exception("No SOI marker")

        while True:
            (marker,) = unpack(">H", data[0:2])

            if marker == 0xFFD8: #Start of Image
                data = data[2:]
            elif marker == 0xFFD9: #End of Image - should not be reached
                return
            else:
                (len_chunk,) = unpack(">H", data[2:4])

                if marker == 0xFFC4: #Huffman Table(s)
                    len_chunk, chunk, data = self.checkHtStructType(len_chunk, data)
                    (marker,) = unpack(">H", data[0:2])
                    (len_chunk,) = unpack(">H", data[2:4])
                    len_chunk += 2
                else:
                    len_chunk += 2
                    chunk = data[4:len_chunk]

                if marker == 0xFFDB: #Quantization Table(s)
                    chunk1 = chunk[:65]
                    self.DefineQuantizationTables(chunk1)
                    #If length is more than 69, only 1 marker for all tables, keep processing
                    if (len_chunk > 69):
                        chunk2 = chunk[65:130]
                        self.DefineQuantizationTables(chunk2)
                elif marker == 0xFFC0: #Start of Frame
                    self.BaselineDCT(chunk)
                elif marker == 0xFFDA: #Start of Scan
                    mcuList = self.StartOfScan(data, len_chunk)
                    return mcuList, self.height, self.width

                data = data[len_chunk:]
            if len(data) == 0:
                return

#Entrypoint function
def extract(inFile):
    img = JPEG(inFile)
    try:
        coeffs, y, x = img.decode()
    except Exception as inst:
        m, = inst.args
        raise Exception("Unexpected error:", m)
    return coeffs, y, x
```

jsdec.py

```
#Custom detector module for JRevealPeg (jrpeg.py), target: Jsteg v. 0.3.0
#Author: Charles Badami

#jsdec.py - Jsteg decoder; looks for magic word, "jsteg", embedded at beginning of hidden data (stego); no pw or encryption
#----- Bits of each "jsteg" byte are stored in reverse order, must put in correct order when retrieving
#----- If bytes decode to "jsteg", Jsteg detected

# #Jsteg steganography characteristics
#Skips -1, 0, 1, even bytes decode to 0, odd to 1
#Only uses ac's in Y components (zig-zag order, edge padding included)

''' 'Magic' key for Jsteg
    01101010  j
    01110011  s
    01110100  t
    01100101  e
    01100111  g
'''


#Bit order is reversed for 'jsteg' magic chars
jstegKey = [
                [0,1,0,1,0,1,1,0],
                [1,1,0,0,1,1,1,0],
                [0,0,1,0,1,1,1,0],
                [1,0,1,0,0,1,1,0],
                [1,1,1,0,0,1,1,0]
            ]


#Filter out DC coeffs, -1, 0, 1; delete Cb, Cr components (every 5th and 6th sublist)
def filterMCUs(coeffs):
    res = []
    count = 1
    for c in coeffs:
        if ((count + 1) % 6 == 0 or count % 6 == 0):
            count += 1
            continue
        c = c[1:] #Ignore DC coeff
        c = list(filter((-1).__ne__, c))
        c = list(filter((0).__ne__, c))
        c = list(filter((0.0).__ne__, c))
        c = list(filter((1).__ne__, c))
        res.extend(c)
        count += 1

    return res

#Compare extracted bits to magic bits
def magic(coeffs, key):
    bitNum = 0
    for i in range(5):
        temp = []
        for j in range(8):
            temp.append(coeffs[bitNum]%2)
            bitNum += 1
        if (temp != key[i]):
            return False

    return True


#Entrypoint function
def detect(coeffs):
    res = False
    try:
        procList = filterMCUs(coeffs)
        res = magic(procList, jstegKey)
    except Exception as inst:
        m, = inst.args
        raise Exception("Unexpected error:", m)

    return res
```

spdec.py

```
#Custom detector module for JRevealPEG (jrpeg.py), target: SteganPEG v. 1.0
#Author: Charles Badami
'''Credit: Some of this code is the result of adapting/revising source code and transcribing to Python from Visual Basic,
    referencing SteganPEG 1.0 by Kango Abhiram, https://www.softpedia.com/get/Security/Encrypting/SteganPEG.shtml'''

#spdec.py - SteganPEG decoder; pw and encryption employed; assume pw is from a list of common/weak passwords;
#----- Need to encode pw, use to decrypt header (first 4 bytes of stego)
#----- Get length of (compressed) data from header, retrieve the checksum, which is the byte after the data, decrypt
#----- Decrypt data using sp algo, calculate checksum using checkval algo
#----- Compare calculated checksum to retrieved value; if they match, SteganPEG detected

# #SteganPEG steganography characteristics
#Skips 0; negative odds and positive evens decode to 0, else 1
#Uses ac's only, Y's, Cb, Cr (zig-zag order, edge padding included)

#Might need to account for the possibility at least one incomplete MCU at the end of extracted input (reason unknown)
#----- Probably won't need to worry about it, just make sure any loops are flexible about it
#----- Hoping it's an occasional one-off error in SteganPEG

#Most common passwords (top 20) of 2020, according to NordPass
#----- Retrieved 12/2020 from https://en.wikipedia.org/wiki/List_of_the_most_common_passwords
pwds = ['123456',
        '123456789',
        'picture1',
        'password',
        '12345678',
        '111111',
        '123123',
        '12345',
        '1234567890',
        'senha',
        '1234567',
        'qwerty',
        'abc123',
        'Million2',
        '000000',
        '1234',
        'iloveyou',
        'aaron431',
        'password1',
        'qqww1122']


#Filter out DC coeffs, 0
def filterMCUs(coeffs):
    res = []
    for c in coeffs:
        c = c[1:] #Ignore DC coeff
        c = list(filter((0).__ne__, c))
        c = list(filter((0.0).__ne__, c))
        res.extend(c)

    return res


#Reconstruct data bytes from LSB's
def dataBytes(bList):
    byteProcd = 0
    res = []
    count = 1
    for b in bList:
        bit = b % 2
        if (b < 0):
            bit = (~bit) & 1 #Reverse bit if negative
        byteProcd = (byteProcd << 1) + bit
        if (count == 8):
            res.append(byteProcd)
            byteProcd = 0
            count = 0
        count += 1

    return res


#Password encoding algorithm from SteganPEG
def encodePass(pw):
    passStore = []
    passInd = 0
    numBits = 0
    numToAdd = 0
    while (passInd < len(pw)):
        passBits = 0
        passPres = ord(pw[passInd])
        while (passBits < 8):
            while (numBits < 3):
                numToAdd = (numToAdd << 1) | (passPres >> 7)
                numBits += 1
                passBits += 1
                passPres = (passPres << 1) & 0xFF
                if (passBits == 8):
                    break
            if (passBits == 8):
                break
            passStore.append(numToAdd)
            numToAdd = 0
```

```
            numBits = 0
        passInd += 1
    if (numBits != 0):
        passStore.append(numToAdd)
    passStore.reverse()
    return passStore


#RotateLeft (part of decryption)
def rotateLeft(val):
    msb = val >> 7
    newVal = val << 1
    val = newVal & 0xFF
    val = val | msb
    return val


#General decryption sequence, returns a list of decrypted data bytes
def decryptData(byteList, passStore, rotChosen):
    i = 0
    result = []
    while (i < len(byteList)):
        val = byteList[i]
        if (rotChosen == len(passStore)):
            rotChosen = 0
        for j in range(0,passStore[rotChosen]):
            val = rotateLeft(val)
        rotChosen += 1
        result.append(val)
        i += 1

    return result


#Calculate checksum for verification
def calcChecksum(data): #Pass in embed data without 4-byte length header and checksum
    checksum = 0
    sum = 0
    val = 0
    for i in range(len(data)):
        val = (val << 1) | (data[i] & 1)
        if (i % 7 == 0):
            sum += val
            val = 0
    if (val != 0):
        sum += val
        val = 0
    for j in range(8):
        checksum = (checksum << 1) | (sum & 1)
        sum >>= 1

    return checksum
```

```


#Entrypoint function
def detect(coeffs):
    #For logging
    msgs = ""

    #Filter unused bytes
    bList = filterMCUs(coeffs)

    #Reconstruct data bytes
    bList = dataBytes(bList)

    #Encode common passwords
    passStores = []
    for pw in pwds:
        passStores.append(encodePass(pw))

    #Loop through common passwords, return True if checksum verifies
    for i in range(len(pwds)):
        msgs += "Checking password '" + pwds[i] + "'... "

        #Decrypt first four data bytes, extract length of main embedded data
        header = decryptData(bList[:4], passStores[i], 0)
        dataLen = 0
        for b in header:
            dataLen <<= 8
            dataLen = dataLen | b

        #If decoded length is more than (bList-4), data size too big, go to next pw
        if (dataLen > (len(bList) - 4)):
            msgs += "Data length too big, negative match...\n"
            continue

        bList2 = bList[4:dataLen+4] #Trim header and excess bytes
        rot = 4 #Starting rotation index after header decrypted

        decData = decryptData(bList2, passStores[i], rot)
        checksum1 = decData.pop()

        #Validate checksum
        checksum2 = calcChecksum(decData)
        if (checksum1 == checksum2):
            msgs += "Positive match!\n"
            return True, msgs

        msgs += "Negative match...\n"

    return False, msgs
```

ogdec.py

```python
#Custom detector module for JRevealPEG (jrpeg.py), target: OutGuess 0.2.2
#Author: Charles Badami

#ogdec.py - OutGuess decoder; key (password) and encryption employed; assume default options/key is used ("Default key")
#----- OutGuess uses pseudo-random number stream iterator (ARC4) for encryption;
#---------- First 32 iterator values for default key have been extracted for use in detection
#---------- Need to extract "bitmap" bytes before detection (below)
#----- Heuristic detection strategy:
#---------- 1. Extract seed and original embedded data length from header bytes (first 4), using "default" iterator values
#---------- 2. Is original data length > (bitmap length)/2 ? If so, OutGuess not possible, maximum message size exceeded
#---------- 3. Is seed >= 0 and <=255? If not, OutGuess not detected (defaults assumed)

# #OutGuess steganography characteristics
#Skips 1, 0, evens decode to 0, odds 1
#Uses BOTH dc's and ac's, Y's, Cb, Cr (natural order, no edge padding)
##########################################################################################################################################

#Put coeff matrices in natural order
def dezig(coeffs):

    natOrder = [0, 1, 8, 16, 9, 2, 3, 10,
                17, 24, 32, 25, 18, 11, 4, 5,
                12, 19, 26, 33, 40, 48, 41, 34,
                27, 20, 13, 6, 7, 14, 21, 28,
                35, 42, 49, 56, 57, 50, 43, 36,
                29, 22, 15, 23, 30, 37, 44, 51,
                58, 59, 52, 45, 38, 31, 39, 46,
                53, 60, 61, 54, 47, 55, 62, 63]

    dezigged = []
    for c in coeffs:
        output = [0] * 64
        for i in range(64):
            output[natOrder[i]] = int(c[i])
        dezigged.append(output)

    return dezigged


#Calculate indexes of edge MCUs (right, bottom, bottom-right corner)
def calcEdges(y, x): #y is image height, x is width

    #Account for divisible lengths
    if (x % 16 == 0):
        xEdge = x // 16 - 1
    else:
        xEdge = x // 16

    if (y % 16 == 0):
        yEdge = y // 16 - 1
    else:
        yEdge = y // 16


    #Do right side list
    rSide = []
    rStart = xEdge
    rStep = xEdge + 1
    rStop = rStep * yEdge
    if (x % 16 != 0):
        for i in range(rStart, rStop, rStep):
            rSide.append(i)

    #Do bottom list
    bSide = []
    bStart = (xEdge + 1) * yEdge
    bStop = bStart + xEdge
    if (y % 16 != 0):
        for i in range(bStart, bStop):
            bSide.append(i)

    #Do bottom-right corner
    brCorner = [bStart + rStart]

    Incomps = [rSide, bSide, brCorner]

    return Incomps
```

```
#Trim MCU padding from edges, return single list of data bytes
def trimEdges(mcuList, incomps, y, x):

    #Default: Keep bottom-right Y's? Y0 always kept
    brY1 = brY2 = brY3 = True

    #Edit rSide if necessary
    xRem = x % 16
    if (xRem != 0):
        if (xRem <= 8):
            off = 0
            pSave = xRem
            brY1 = brY3 = False
        else:
            off = 1
            pSave = xRem - 8

        #Calc pixels to keep
        leftKeep = []
        for row in range(0, 57, 8):
            for col in range(row, pSave + row):
                leftKeep.append(col)

        #Keep relevant pixels
        for mcu in incomps[0]:
            ylistA = []
            ylistB = []

            for c in range(len(mcuList[mcu*6 + off])):
                if (c in leftKeep):
                    ylistA.append(mcuList[mcu*6 + off][c])
            mcuList[mcu*6 + off] = ylistA

            for c in range(len(mcuList[mcu*6 + off + 2])):
                if (c in leftKeep):
                    ylistB.append(mcuList[mcu*6 + off + 2][c])
            mcuList[mcu*6 + off + 2] = ylistB

    #Edit bSide if necessary
    yRem = y % 16
    if (yRem != 0):
        if (yRem <= 8):
            off = 0
            pSave = yRem
            brY2 = brY3 = False
        else:
            off = 2
            pSave = yRem - 8

        #Calc pixels to keep
        topKeep = []
        for row in range(pSave):
            for col in range(row, row + 8):
                topKeep.append(col)

        #Keep relevant pixels
        for mcu in incomps[1]:
            ylistA = []
            ylistB = []

            for c in range(len(mcuList[mcu*6 + off])):
                if (c in topKeep):
                    ylistA.append(mcuList[mcu*6 + off][c])
            mcuList[mcu*6 + off] = ylistA

            for c in range(len(mcuList[mcu*6 + off + 1])):
                if (c in topKeep):
                    ylistB.append(mcuList[mcu*6 + off + 1][c])
            mcuList[mcu*6 + off + 1] = ylistB
```

```
    #Edit brCorner
    brY0Keep = []
    brY1Keep = []
    brY2Keep = []
    brY3Keep = []

    if (xRem % 8 == 0):
        rLimit = 8
    else:
        rLimit = xRem % 8
    if (yRem % 8 == 0):
        bLimit = 8
    else:
        bLimit = yRem % 8

    for row in range(8):
        for a in range(row * 8, 8):
            if (a < xRem and (row * 8) < yRem):
                brY0Keep.append(a)
            if (brY1):
                if (a < rLimit and (row * 8) < bLimit):
                    brY1Keep.append(a)
            if (brY2):
                if (a < xRem and (row * 8) < bLimit):
                    brY2Keep.append(a)
            if (brY3):
                if (a < rLimit and (row * 8) < bLimit):
                    brY3Keep.append(a)

    mcu = incomps[2][0]
    brList = []
    for c in range(64):
        if (c in brY0Keep):
            brList.append(mcuList[mcu*6][c])
    if (brY1):
        for c in range(64):
            if (c in brY1Keep):
                brList.append(mcuList[mcu*6 + 1][c])
    if (brY2):
        for c in range(64):
            if (c in brY2Keep):
                brList.append(mcuList[mcu*6 + 2][c])
    if (brY3):
        for c in range(64):
            if (c in brY3Keep):
                brList.append(mcuList[mcu*6 + 3][c])

    mcuList[mcu*6] = brList
```

```
    #Delete unnecessary Y components
    #rSide is incomps[0]; bSide is incomps[1]; brCorner is incomps[2]
    #Y's are mcu*6 + <offset>, i.e. 0-3 for Y0-Y3; 4-5 for Cb, Cr
    temp = []
    for comp in range(0, len(mcuList), 6):
        mcu = comp // 6
        off = comp % 6
        if (mcu in incomps[0]): #Keep Y1, Y3 if necessary
            temp.append(mcuList[mcu*6])
            if (brY1):
                temp.append(mcuList[mcu*6 + 1])
            temp.append(mcuList[mcu*6 + 2])
            if (brY3):
                temp.append(mcuList[mcu*6 + 3])
            temp.append(mcuList[mcu*6 + 4])
            temp.append(mcuList[mcu*6 + 5])
        elif (mcu in incomps[1]): #Keep Y2, Y3 if necessary
            temp.append(mcuList[mcu*6])
            temp.append(mcuList[mcu*6 + 1])
            if (brY2):
                temp.append(mcuList[mcu*6 + 2])
            if (brY3):
                temp.append(mcuList[mcu*6 + 3])
            temp.append(mcuList[mcu*6 + 4])
            temp.append(mcuList[mcu*6 + 5])
        elif (mcu in incomps[2]): #Keep Y1, Y2, Y3 if necessary
            temp.append(mcuList[mcu*6])
            if (brY1):
                temp.append(mcuList[mcu*6 + 1])
            if (brY2):
                temp.append(mcuList[mcu*6 + 2])
            if (brY3):
                temp.append(mcuList[mcu*6 + 3])
            temp.append(mcuList[mcu*6 + 4])
            temp.append(mcuList[mcu*6 + 5])
        #Else add all Y's and Cb, Cr
        else:
            temp.append(mcuList[mcu*6])
            temp.append(mcuList[mcu*6 + 1])
            temp.append(mcuList[mcu*6 + 2])
            temp.append(mcuList[mcu*6 + 3])
            temp.append(mcuList[mcu*6 + 4])
            temp.append(mcuList[mcu*6 + 5])

    mcuList = temp

    return mcuList
```

```
#Turn data bytes into LSB's (even is 0, odd is 1)
def bitmap(dataBytes):
    bitmap = []
    for element in dataBytes:
        if (element % 2 == 0):
            bitmap.append(0)
        else:
            bitmap.append(1)

    return bitmap


#Extract random seed and data length from beginning of data
def extractHeaderInfo(data):

    #Pre-calculated iterator and encryption key values for "Default key"
    iterator = [29, 50, 70, 71, 86, 93, 125, 140,
            167, 172, 194, 209, 233, 238, 264, 294,
            308, 313, 326, 336, 362, 388, 402, 413,
            426, 452, 463, 495, 503, 512, 520, 526]

    arc4key = [0x3f, 0x5e, 0xd7, 0x1c]

    #Reconstruct 4 header bytes (encrypted)
    encBytes = []
    for i in range(7,32,8):
        byte = 0
        off = 7
        for j in range(i,i-8,-1):
            byte += data[iterator[j]] << off
            off -= 1
        encBytes.append(byte)

    #Decrypt header
    hdr = []
    for i in range(4):
        hdr.append(encBytes[i] ^ arc4key[i])

    #Calc seed and data length
    seed = (hdr[1] << 8) + hdr[0]
    dataLen = (hdr[3] << 8) + hdr[2]

    return seed, dataLen


#Entrypoint function
def detect(coeffs, y, x):
    #For logging
    msgs = ""

    #Put coeffs in natural order
    dezigged = dezig(coeffs)
    mcus = dezigged

    #If height or width is not divisibly by 16, need to trim padding
    if (y % 16 != 0 or x % 16 != 0):

        #Calculate edges
        incomps = calcEdges(y, x)

        #Delete padding, extra components
        try:
            mcus = trimEdges(dezigged, incomps, y, x)
        except Exception as inst:
            m, = inst.args
            raise Exception("Unexpected error:", m)

    #Now filter out 1's and 0's and save bytes in single list
    bitmapBytes = []
    for line in mcus:
        line = list(filter((0).__ne__, line))
        line = list(filter((1).__ne__, line))
        bitmapBytes += line

    #Extract LSB's
    bmap = bitmap(bitmapBytes)

    #Detection heuristics
    seed, dataLen = extractHeaderInfo(bmap)

    #OutGuess sets this data size limit
    if (dataLen > len(bmap) // 2):
        msgs += "Data length too big vs. available cover..."
        return False, msgs
    #Seed should be 255 or less for default key
    if (seed > 255):
        msgs += "Seed too large..."
        return False, msgs

    #If header doesn't rule it out, assume OutGuess possible
    return True, msgs
```

APPENDIX D

LINKS TO FREE JPEG IMAGES USED

https://unsplash.com/photos/BOuggN1tMEk
https://unsplash.com/photos/LMU2w-K4J7k
https://unsplash.com/photos/0O4C9qtn_48
https://unsplash.com/photos/p9t7g5ORALs
https://www.freeimages.com/photo/houseplant-1640441
https://www.freeimages.com/photo/palm-and-pier-1390929
https://unsplash.com/photos/DE6yhZyG8bE
https://www.freeimages.com/photo/pumpkins-1363061
https://unsplash.com/photos/Bk5sT_CzOaQ
https://www.freeimages.com/photo/yellow-1640997
https://unsplash.com/photos/K2DT1Z_bthw
https://unsplash.com/photos/BUzjetL88RU
https://unsplash.com/photos/I8O58UfInLc
https://www.freeimages.com/photo/fall-in-qom-province-1640647
https://unsplash.com/photos/I9KqhcDU34c
https://unsplash.com/photos/coUZnech6qw
https://unsplash.com/photos/2BbwrlmIaX8
https://unsplash.com/photos/WZXROzZpftw
https://www.freeimages.com/photo/bengal-tiger-1521311
https://www.freeimages.com/photo/white-house-1221438