stegSecure: Final Report

Authors: Steve Huang, Asher Kang, Maria Ringes, David Shen*

Table of Contents

I. Introduction

Problem

Steganography is the practice of hiding a secret message in something that is not secret without arousing any suspicion about the existence of a hidden message. Technologically speaking, steganography refers to the hiding of information, files, or even programs in seemingly benign "cover images" such as files, images, videos, and even network packets. A common steganography technique is known as Least Significant Bit (LSB) steganography. This technique encodes the binary representation of the hidden data in the least significant bits of an image, causing minimal alteration to the cover image and escaping detection from the human eye.

The use of steganography to execute malicious attacks and to spread messages with malicious intent is on the rise. Today, with the rise of social media, millions of image files are uploaded and exchanged across the internet, with each file having the potential to carry malicious data via LSB steganography. Despite the fact that many users have installed anti-virus softwares, most infected files pass through these installed anti-virus softwares because the cover images are deemed as benign. Fortunately, there do exist steganalysis tools that identify steganographic content within a file. However, most of these tools require a user to have an initial suspicion about some file, and manually input the file into the tool to be scanned. Moreover, even if an image is proven to be steganographic, these steganalysis tools do not often sanitize the image afterwards, erasing the potentially malicious hidden information. File sanitation tools exist that perform this job, but they often exist as a standalone tool. Overall, there exist several tools that tackle partial fragments of the crisis of malicious steganography, but there is yet a single software that brings their combined power together.

Solution

stegSecure is a software that intercepts downloaded user images, detects LSB-steganography in images among these user files, and sanitizes them of hidden information. Antivirus softwares notoriously fail to detect malicious code that is hidden inside benign "cover images." Our software intends to inspect a newly downloaded image for steganographically hidden messages in the image. stegSecure fulfills a critical need in cyber security because it introduces a new layer of cyber protection for a user without requiring a user to directly interact with a system. For the purposes of this class, this version of stegSecure only focuses on LSB-steganography, the most ubiquitous form of steganography. With more time and resources, this software could be expanded to all file types and several more steganography techniques.

The pathway to building stegSecure consisted of three phases. The first phase was to implement a command-line LSB steganography tool that allows users to hide a message inside a cover image. A secondary feature of this tool allows users to reverse engineer the steganography and

extract the hidden message when given a steganographic image. The second phase was to implement LSB steganalysis, which allows the program to successfully judge whether a given image is carrying steganographically hidden data. The third and final phase combined the above features with a file sanitation feature as a unified steganography detection and file sanitation software.

Table of Product Features

Features	**Steganalysis Tools (e.g. stegdetect)**	**Anti-Virus Tools (e.g. Kaspersky)**	**File sanitization (e.g. Votiro)**	**stegSecure**
Detection of steganographic content in one input image	✓	X	X	✓
Apply detection of steganographic content to newly saved images	X	X	X	✓
Prevent steganographic viruses by automatically sanitizing the image	X	X	✓	✓
Useful for consumer devices	✓	✓	X	✓

Github: https://github.com/standardrhyme/stegsecure

II. Video

https://app.animaker.com/animo/FUKi31qwuCiKzyOI/

III. Technical Specifications

Architecture Overview

stegSecure involves 3 main layers:

- The **interceptionfs** layer is implemented in pkg/interceptionfs and is responsible for:
    - Initializing a virtual filesystem with passthrough capability in memory
    - Mounting the virtual filesystem in the specified directory
    - Passing modified file data to the steganalysis algorithms
    - Blocking read access to unscanned files
    - Sanitizing potentially steganographic files
    - Releasing unsteganographic and sanitized files back to the underlying filesystem
- The **steganalysis** layer is implemented in pkg/steganalysis and is responsible for:
    - Analyzing files for potential steganographic content
    - Sending any files that need sanitation to the sanitize layer
    - Passing the resulting file data back to the interceptionfs layer
- The **sanitize** layer is implemented in pkg/sanitize and is responsible for:
    - Removing any steganographic content from files

The interceptionfs layer receives a notifier function when initialized, which is called with any modified Nodes. The interceptionfs layer implements the following functions:

- `func Init(notifier func(Node)) (*FS, error)`: Initialize a filesystem and data structures needed, as well as a root directory.
- `func (f *FS) Mount(mountpoint string) error`: Create a new fuse connection at the specified mountpont, and bind mount it to a temporary directory.
- `func (f *FS) Serve(res chan error) error`: Connect the filesystem to the fuse connection.
- `func (f *FS) Close() error`: Close the fuse connection, unmount the bind mount, and delete the temporary directory.

Once a file is modified, it calls its notifier function, which is the steganalysis analyze function. The steganalysis layer implements the following function:

- `func Analyze(n interceptionfs.Node)`: Analyze the node for any potentially steganographic content, and send it to the sanitize layer if necessary.

The sanitize layer implements the following function:

- `func Sanitize(n interceptionfs.Node)`: Remove any potentially steganographic content from the node.

Custom Data Structures

FS
A FS struct (fs.go) represents an interception filesystem, and contains its child nodes as well as the mount configuration and the next available inodes.

Node
The Node interface (node.go) represents the shared methods between Dir and File.

NodeAttr
A NodeAttr struct (node.go) holds the attributes of each node, including file size, access, modification, and creation times, file mode, and file owner.

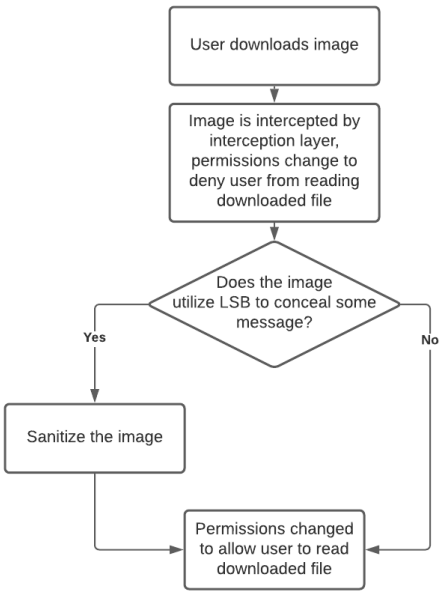
Dir
A Dir struct (dir.go) represents a directory in the interception filesystem, and contains its parent directory and child nodes.

File
A File struct (file.go) represents a file in the interception filesystem, including the file data and its parent directory.

FileHandle
A FileHandle struct (filehandle.go) is used to access the contents of a file. A passthrough handle contains an os filehandle, while a non-passthrough handle blocks read access to files still being processed. It also sends the file to the steganalysis layer after it has been modified.

The InterceptionFS Layer

The interceptionfs layer creates and mounts a virtual filesystem over the user's Downloads directory. It allows for two types of files and directories: virtual ones, which are access-controlled by stegSecure, and passthrough files and directories, which exist in the underlying Downloads folder. Read access is prevented at the filesystem level for any virtual files, so even programs with root permissions will not be able to read the file. Write access is allowed, and calls the DebouncedNotifier for the file with each modification. This waits until the file has not been modified for a second, before sending it to the steganalysis layer.



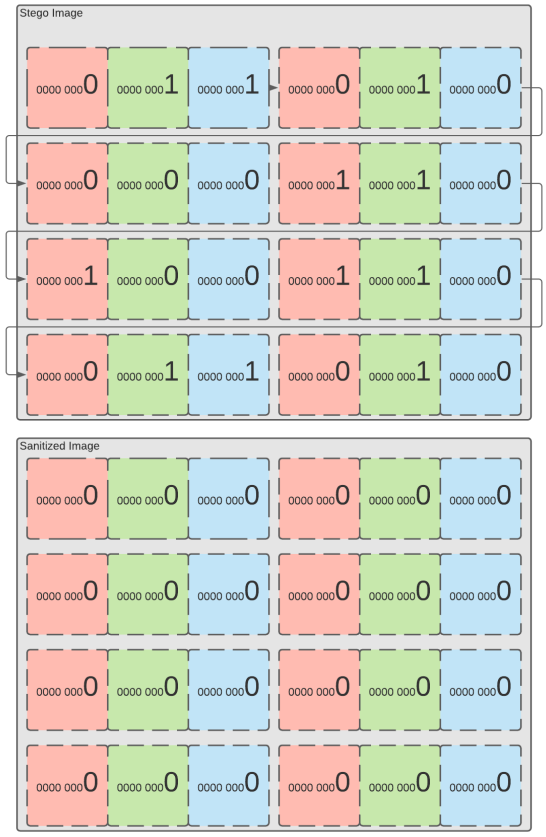
File Interception Architecture

The Steganalysis Layer

The steganalysis layer uses the sample pairs algorithm, written in Go, to check for potentially steganographic content. It receives a Node from the interceptionfs modification notifier and uses that to retrieve the bytes of the image. This is something that can only be done within stegSecure, because the file contents never leave the program's memory while being processed. It then uses the image.Decode function from the image package to convert the bytes into an image, before running the sample pairs analysis on it. If it determines that there is steganographic content, it then sends it to the sanitize layer.

The Sanitize Layer

The sanitize layer removes any potential steganographic content, encoded using the LSB method, from images. After receiving the bytes, it converts it into an image before setting each least significant bit of each pixel's color channels to zero. This overwrites any potential steganographic content and sets it to zeros.



File Sanitation Bit Masking

Tech Stack

The initial prototype for stegSecure was built with Python. We chose to use Python for the command-line LSB steganography tool, the reverse steganography, steganalysis, and file sanitation features that comprise the prototype. We chose Python because of the quality of image processing packages available in Python. Specifically, we implemented Pillow, which allows us to process image data that is stored in pixel format. Our steganography, reverse steganography, and steganalysis implementations break down and analyze the binary RGB pixel values of images. While many languages have adequate image processing libraries, we decided to use Python for this portion of our project, because the goal was to learn about LSB steganography and image formats. Python is a suitable language for quick prototyping.

We chose to use Go for the overarching architecture and final version of stegSecure. Go is efficient in terms of energy, time, and memory consumption, especially when compared to Python outside of image processing tasks. Because we would like stegSecure to cause minimal delay to the user's downloading of the image as much as possible, we felt that Go would be the best language to build the final version of stegSecure using Go. Additionally, Go's power to run

concurrent processes also allowed us to implement simultaneous steganalysis of multiple images at once, which is another reason we chose to use it for the final product.

For our virtual file interception feature, we used FUSE, which is a software interface that allows for userspace file systems, which essentially allows us to easily build the interception filesystem without having to touch kernel code.

Design Decisions

A major design decision for stegSecure was choosing to use the Sample Pairs steganalysis algorithm for our steganalysis feature. Among those presented by Boehm (2014) in his survey of major steganalysis algorithms, Sample Pairs was the best documented and straightforward. The others, including Chi-Square Attack (Westfield 2000) and RS Analysis (Fridrich 2001) involve heavy statistical analysis and the implementation of machine learning algorithms. On the other hand, Sample Pairs groups pixel values into pairs, both horizontally and vertically. It forms a finite state machine "whose states are selected multisets of sample pairs called trace multisets" and whose transitions from one state to another come with given probabilities (Dumitrescu 2003). When the Least Significant Bits have been altered via LSB-steganography, the cardinalities of these states change, disrupting the predictable probabilities of state transitions by a measurable amount. The amount of disruption is correlated with the size of the hidden data that is manipulated onto the LSBs of the image. This magnitude is approximated via a quadratic formula, and ultimately represents the probability that the image is steganographic (Dumitrescu 2003).

For our file interception feature of stegSecure, we decided to prevent the reader from reading the file while stegSecure is scanning the image. This is because in the real world, the malicious impact of the steganographic image is realized when the image is actually opened by the user. By preventing the user from opening and reading the file after executing the download command, stegSecure has a chance to examine and sanitize the file of hidden, potentially harmful information.

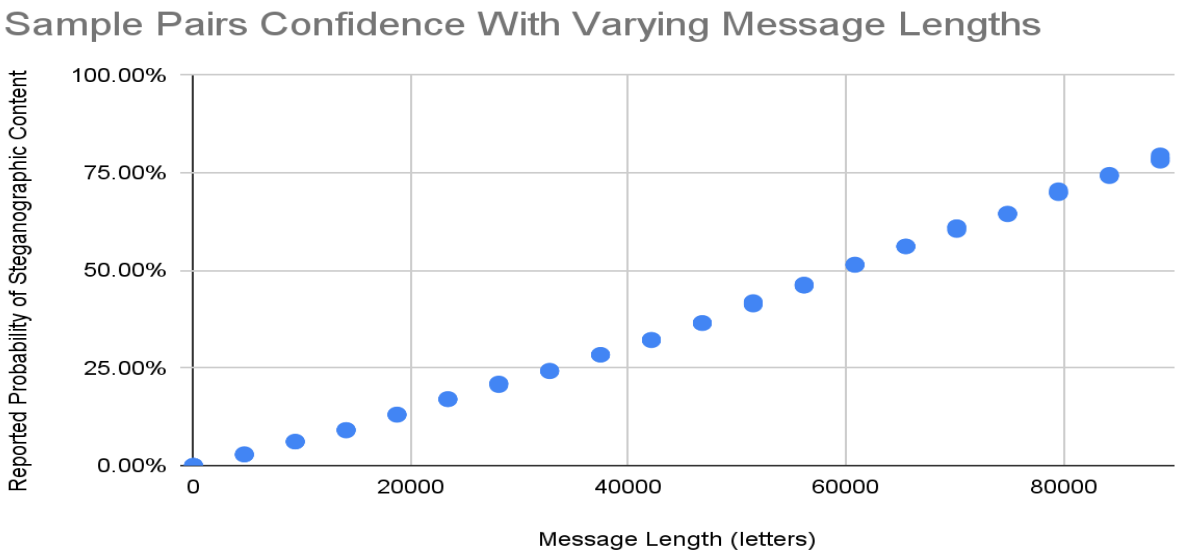
Potential Limitations

Different image formats are encoded differently. And our product must handle each image format with separate code. Although Go has a comprehensive image package, an attacker can still hypothetically utilize an image format that is not frequently used, or even invent a completely new image format specifically for steganography. Within the scope of popular image format, we have handled both PNG and JPEG images in this project, but we have not yet covered GIF images for the sake of time.

Our project also requires the steganographic image to be actively downloaded by the users. It can only identify and sanitize steganography that will potentially be stored in the hard drive of the users' machines. If the malicious code in an image can be activated when the user simply accesses a browser page with the steganography, our product cannot stop it. Besides, it is not feasible to do a steganalysis on every image in a browser page whenever the user opens a new page.

Performance

We measured our steganalysis tool's ability to detect steganographic content in varying message lengths. In our experiments, we used a single 360,780 byte image, and 5 random trials for message lengths, increasing from 10 to 25% of the image size. With 50% as our minimum probability to consider an image steganographic, the smallest detected message was over 60,000 letters long. This consists of about 17% of the entire file content.



Running samplesmaller.png (353 KB) and samplestego.png (300 KB) through stegSecure took 1.06s and 1.34s, respectively. samplestego.png took longer, because stegSecure had to sanitize the image as well. This gives us a speed of around 3 ms/KB to process images without steganographic content, and 4.4 ms/KB to analyze and sanitize images with steganographic content. This includes the time needed to move the file from the interception filesystem to the underlying filesystem, and change it to a passthrough file.

IV. Project Milestones

Our initial report was framed with several milestones. These milestones, as detailed in our initial report, fell into three general categories: steganography, steganalysis, and their combination with a file interception system. Overall, we successfully met all of our milestones, affirmed in these three fully functioning components of our final stegSecure software.

Phase One: Implement Command-Line Steganography

Our initial report stated, “the program will receive a user-input cover image and message as input, and return the LSB-steganographic image concealing the existence of the specific hidden data as output.” Although this code was not directly used in our final source code, the lessons learned from such implementation allowed us to better understand how our final project should be framed. Furthermore, this command line tool allowed us to create our own steganographic images that we could test against our final product.

Our initial report also stated, “the command line implementation will also have the ability to reverse the LSB steganography process, and extract the hidden data from the cover image”. This milestone, too, was completed. Again, this code was not directly used in our final source code, however our demonstration utilized such to prove to viewers exactly what our program was capable of.

Phase Two: Implement LSB Steganalysis

Our initial report stated, “this project will implement LSB steganalysis, which is the detection of the presence of hidden data in a given file”. To complete such a milestone, we first created a python script that would allow users to interact via the command line. This source code was altered slightly to fit in with our final implementation.

Phase Three: stegSecure Software

Our initial report stated, “this project will utilize the previously explained features to form a steganography identification and sanitation software. When a user downloads a JPEG image, it will first be saved to a controlled environment, such that if the message is malicious, the system will not be impacted.” This milestone was also achieved. When writing our initial report, we were unsure as to what issues we would run into had we chosen to aim for a file interception layer. With such, we aimed to achieve a controlled environment. As our project progressed, we decided it would be more secure to intercept the file. Thus, we implemented the file interception layer seen in our final implementation.

V. Conclusions

What did we learn?

Ultimately, we learned much about the practice of steganography and steganalysis, its uses, its mechanisms, and its implementations. Specifically, we learned about the most ubiquitous form of steganography–Least Significant Bit steganography. In doing so, we learned about the composition of image files, and how pixels are arranged by bit values. We dove deep into how the Sample Pairs steganalysis algorithm works, as well as examining the mechanisms of other steganalysis algorithms. It was by no means easy to understand any of these algorithms, and Sample Pairs was the easiest to understand, because it is based on set theory, which we are all familiar with. It was also more well documented than the other algorithms, so we were able to understand its use better. In implementing our own version of the Sample Pairs algorithm, first in Python, and later in Go, we learned how to use the standard image packages for both of the languages–PIL for Python, and Images for Go.

In the broader scheme, we also gained valuable experience starting a project from scratch, including researching, designing, asking, and implementing the product, all with a team of people with different backgrounds and skill sets. Certainly, our group came equipped with a wide range of expertise, such that we could come together to implement a variety of features into this single product. Some had experience working with file interception and moving, some had experience prototyping and making software engineering products, some were good at presenting information, and others were good at making sure reports flowed well and presented a clear, coherent story. There was also the challenge of setting our own milestones and meeting our deadlines, which we found challenging at times, yet still fulfilled to the very end.

Do we agree to share our code/report/video with other students?

Yes

Future Suggestions

One suggestion is to strictly limit the scope of the final project to Distributed Systems. Our team ended up choosing a topic that, while related to the field of Cyber Security, did not strongly relate to our class materials. As such, it oftentimes felt like we were building a product without directly applying the lessons learned in your class. It did not feel like much of a final project for your class as it did a general CS project. Perhaps as the instructor of the course, you can help future students channel their knowledge of distributed systems, consensus, blockchain, etc. into a more precise distributed systems project. This way, students can more concretely and securely get hands-on experience with system design and distributed systems, while demonstrating all that they learned through your course over the semester. This would make the final project more relevant to our studies of the semester and a rewarding conclusion to Everything Distributed.

Moreover, it would help us to concretely understand and visualize distributed systems at work in real-world contexts.

VI. References

Boehm, B. (2014). Stegexpose-A tool for detecting LSB steganography. *arXiv preprint arXiv:1410.6656*.

Dumitrescu, Sorina, Xiaolin Wu, and Zhe Wang (2003). "Detection of LSB steganography via sample pair analysis". In: Signal Processing, IEEE Transactions on 51.7, pp. 1995–2007.

Fridrich, Jessica, Miroslav Goljan, and Rui Du (2001). "Reliable detection of LSB steganography in color and grayscale images". In: Proceedings of the 2001 workshop on Multimedia and security: new challenges. ACM, pp. 27–30.

Westfeld, Andreas and Andreas Pfitzmann (2000). "Attacks on steganographic systems". In: Information Hiding. Springer, pp. 61–76