Steganography

Manu Adithyan

June 2024

Introduction

This report discusses about the implementation of a Steganography tool in Go and experiences from this implementation.

Which type of files to use

The essence of steganography is to hide data in a file in a way that does not alter the original file noticeably. There are a lot of options for this. For instance it can be an image file or an audio file. The author implemented a tool that hides data in image files of PNG format. PNG format was used due to it's lossless compression and ubiquity. Other formats like BMP or TIFF are feasible but are less common for images and does not provide compression. JPG is not feasible due to lossy compression. Lossy compression can alter LSBs of color levels quite a bit and the data is hidden in the LSBs of color levels of the image.

Why Go

This project was implemented in Go rather than in other suggested languages like Java, Python, C, or C++ due to several key advantages that Go offers, particularly in the context of this project:

- **Reduced Complexity:** Go's simple and minimalistic syntax is easier from a developer perspective, facilitating faster development and readability.
- **Superior Concurrency Model:** Go's lightweight and efficient goroutines and channels which are used in this project extensively enable better scalability and ease of implementation compared to thread-based concurrency´.
- **Enhanced Performance:** Go compiles to native machine code, resulting in significantly faster execution times and improved runtime performance.

- **Lower Overhead:** Statically linked binaries and the absence of a virtual machine reduce runtime overhead, leading to faster startup times and lower memory usage.

Metadata

When hiding data in a target file it is necessary to store metadata of the file so that it can be extracted and recovered from the target file. The author's solution to this was a 64 byte header. Bytes 0 to 3 represent the size of the file in Big Endian format. Since 4 bytes can represent up to 4.29E9 it was considered more that satisfactory. The size contains the value of size of the file in bytes if it is not encrypted and as 2's complement of the same if the hidden data is encrypted. Bytes 4 to 35 contains the generated salt for the argon2id hash function and bytes 36 to 47 contains the Initialization Vector for the Rijndael-GCM-seal. If the data is not encrypted these bytes are populated with random values. Bytes 48 to 63 contain the file extension string encoded as a byte. It was required to leave this as 16 bytes due to the block size of Rijndael.

Limit of hiding data in a color level

The author did extensive experiments for a balanced solution of size of data that can be stored and the visisblity of changes in processed images. At the start author worked with the color model RGBA which is 32 bit depth and each color level consist on one byte. In this color model the A or Alpha level cannot be altered as this will cause significant artifacts in the image because alpha determines the transparency of the pixel. With this color model the author had experimented with implementation which hides 1 byte in a pixel (3-2-3) which corresponds to 3 LSBs or R, 2 LSBs of G and 3 LSBs of B. This already caused artefacts in images and the amount of data stored was very less. Hence, it was decided to switch to RGBA64 color model which has 64bit color depth and 48 bit usable excluding alpha. For this color model the author tested hiding 1(3-2-3) to 3(8-8-8) bytes in a pixel. Both of these did not cause any noticeable artefacts for an arbitrary image and the hiding capacity with 3 bytes in a pixel was very good so 3 bytes in one pixel was used as final. Altering more than 8 LSBs will certainly cause artefacts as one color level is 16 bits. It was also noticeable when working with RGBA that artefacts were more visible in images with homogeneous regions and it was almost invisible in heterogeneous regions. The darkness of the image also decreased the visibility of artefacts. But while using RGBA64 this was drastically reduced for all the tested images. The tool converts normal RGBA images to RGBA64 before hiding data. The tested images are provided at the end of the report

How to encode

There are a lot of strategies to encode data into a color level. Any c' such that c' mod 2n = v. One way is to add or subtract the required number to the color level or keep it as it is if it is the same. Another way is to mask every other bit except the required number of LSBs and do a bit wise OR with the required bits from the actual data. The author used the latter due to ease of implementation. In the author's implementation each color level is an unsigned 16 bit integer.The 8 MSBs are masked and preserved and the remaining 8 LSBs contain one byte of the data as it is in Big Endian format. So for RGBA64 if the data is an array of b1,b2 and b3 8 LSBs of R contains b1, 8 LSBs of G contains b2 and so on. it is always better to use the least amount of bits from G then R and then the most from B because changes in G is the most noticeable followed by R and the least noticeable is B. Even with 8 LSBs altered equally in RGBA64 there was no significant artefact formation in the test image which makes RGBA64 very optimal for this. Despite all these positives, one important consideration is that using RGBA64 significantly increases the file size.

Tested Images

The results from tests are provided below for each image the same data file of around 1.1 MB was used with encryption. An extensive inspection with the human eye would find artefacts in images using the RGBA color model which is of depth 32 bit. The RGBA Model stores 1 byte in a pixel in 3-2-3 format. Whereas, RGBA64 model stores 3 bytes in one pixel in 8-8-8 format.

Figure 1: RGBA color model

Figure 2: RGBA64 color model

Figure 3: RGBA color model

Figure 4: RGBA64 color model