Jsteg: A Simple Image Steganography Utility for Data Hiding

Dean Elliot Gramcko
California State University - Sacramento
Sacramento, CA, United State
dgramcko@csus.edu

Jessica Garcia
California State University - Sacramento
Sacramento, CA, United State
meigarcia@csus.edu

Tran Ngoc Bao Huynh
California State University - Sacramento
Sacramento, CA, United State
tranngocbaohuynh@csus.edu

Abstract**—***Steganography is one of the popular technique to hide data and sensitive file into pictures, video, or audio. This paper will present a simple method to implement image steganography utility for data hiding.*

Keywords**—***Steganography, JPEG, hidding data, pictures, hexadecimal trailer.*

I. Introduction

In the Era of Internet of Things, digital data are vulnerability to many ingenious attacks. Not only digital data, but the confidential and authentication of any other type of information and communication are important. This can be observed throughout the development of cryptography, steganography, and variety data hiding techniques. In Cryptography, sensitive data will be encrypted into unreadable text using multiple math-based theory. Meanwhile, in Steganography, the secret message will be disguise into different kind of media files such as images and video. To explore the topic of data hiding forensics, our group will develop a novel hybrid steganography tool which encrypt the secret file and hide it into multiple pictures in an image directory.

II. Background

Image Steganography is a method in which one can hide secret messages in a picture. The picture will not look visually different to the human eye but when compared with a software one can see the difference between the original and the tampered picture. In fact, people who are not in the communication should not realize the tranmission or the exist of hiding data using steganography method [1]. There are different techniques in steganography, such as Least Significant Bit (LSB), Frequency Domain, Transformation, Masking and Filtering technique. [2] [3]

III. Methodology

Our Goal for this project is to find a method to conceal an arbitrary amount of data in payloads of 10's or 100's of MB, inconspicuously. A digital camera SD card would be a generic place to store hidden data, but it offers a variety of cover. Any sized SD card can hold a variety of file types and are capable of holding up to large amounts of data. We want to utilize the space of different files to split and hide a text file in multiple files on the card. We have considered using Least Significant Bit (LSB) steganography strategies as well as hiding data in a file's slack space but these methods would constrain the amount of data that can be stored. To bypass the constraint we settled to add to the tail end of the file.

IV. Code Implementation

In this section, we will discuss our methodology and code implemenetation for the application. The program is build primarily in C. There are six major functions to perform the task: Encrypting the data, Split the data into chunks, Hide chunk in different images, Extract the chunk, Reassemble, and Decrypt to get the data back.

A. Encryption and Decryption Function

The encrypt/decrypt function is built using block cipher in GCM (galois counter mode). We first create functions to mix the byte together. Then, that obfuscation function is used in counter mode with the key to create a cipher text. There is also an initial vector IV or the nonce for GCM. After achieving the encrypted text, we will hash the key, the cipher text and xor with the IV then encrypt the result to create the authentication tag. Finally, the Encryption Function will return the encrypted file which contain the cipher text, the authentication tag, and the IV.

Because of time limited and to make it more simple for this project, we will just use the same SIV (both encrypt and decrypt) (so siv will both authenticates the plaintext and serves as an IV for counter mode encryption). In fact, a practical application should have different IV. When two message encrypted using the same key, if their IV/nonce are repeated (identical) or not used, the data will be leak. Therefore, attackers can perform forgery attack toward the authentication. [4]

In the demo, we used the same key, and not prompt users for key value. However, this can be modified, so that each time before hidding or extracting the data from pictures, users will be asked to input a key or password.

B. Split Function

The split function will divide the hidden data into smaller chunks. It will receive a path to the encrypted text and a path to the pictures directory. . Therefore, the function will first calculate how many picture are there in the directory. Then it will calculate the size of each chunk to hide:

```c
struct dirent* entry;
DIR* dir_path = opendir(jpg_directory);

// count the number of picture in the directory:
while ((entry = readdir(dir_path)) != NULL)
{
    if (entry -> d_type == DT_REG)
    file_count++;
}

// calculate size of each chunk:
size_t chunk_size = ceil(file_size /
↪  (double)(file_count));
```

We will store the data from top to bottom of the directory order. In real practice, it will be better to store the chunk in randomly order, and then have the order saved somewhere or we can have the chunk stores order itself. So later on, the data will be correctly reassemble back. Since the number of byte to store is integer, there will be cases that when we divide the size of file by the number of pictures, the result will not be integer. Therefore we will use ceil function to round up the chunk size. This mean, there probably cases that not all the picture contain hidding data, and some pictures might have less data then the others.

After having the chunk size and everything ready, we will go to each picture in the directory and get its full path. Then we will call the function to hide the chunks into pictures until there are no more data to hide.

```
// split, get chunk, and hide chunk in the jpeg:
while (((entry = readdir(dir_path)) != NULL) &&
↪  (!done))
{

    if (strcmp(entry->d_name, ".") != 0 &&
    ↪  strcmp(entry->d_name, "..") != 0)
    {
        // create new full path
        strcpy(full_path, jpg_directory);
        strcat(full_path, "/");
        strcat(full_path, entry->d_name);

        // if there are still byte to read from
        ↪  encryption file, hide it to the image:
        if (fread(chunk, 1, chunk_size, enc))
        {
            hide_chunk(full_path, chunk,
            ↪  chunk_size);
        }
        else
            done = 1;
    }
}
```

C. Hide Function

Our data hiding strategy leveraged the fact that JPEG images contain a specific hexadecimal trailer (0xFFD9) to indicate the end of their data [5]. Image viewers render the data up to this tag, but don't read data following it, meaning that data appended to the file after that marker does not impact how the image renders in a viewer. Initially we had considered alternate approaches such as hiding data within the slack space of FAT filesystem present on most digital camera memory cards, but that strategy as well as common strategies such as hiding data in the least significant bit(s) of the image data would place undesirable constraints on the amount of data we are able to hide per file when compared with this strategy.

Once the payload has been encrypted and divided into chunks, those chunks are passed to a function with the following parameters: a string containing the path and name of the JPEG file to hide the chunk in, a pointer to the starting address of the chunk to be hidden, and the size of the chunk. With that information, the function calculates the size of the jpeg file, then allocates enough memory to hold the JPEG, the chunk, and 3 additional bytes. Next, the JPEG file is read into the buffer, and then each byte of the JPEG data is iterated over looking for the value 0xFFD9. As the value 0xFFD9 may appear multiple times within the same JPEG file, the offset of this trailer is stored each time it is encountered, overwriting the last one, so the final value stored is the location of the last occurrence of 0xFFD9 in the buffer.

```
int j = 0; //keep track of where we are in the jpeg
↪  data
int end_marker = -1; //the offset of the first free
↪  byte of memory after the final FFD9 (and the
↪  added  0x24 0xFF 0xD9 trailer we put in)

//copy all of the data from the jpeg into the buffer
fread(data_to_write, 1, len, jpg);

for(j; j < len; j++)
{
    if(j > 0)
    {
        if(data_to_write[j-1] == 0xff &&
        ↪  data_to_write[j] == 0xd9) //  make note
        ↪  every time we find FFD9
        {
            end_marker = j; // store as end_marker
            ↪  until we find the last one
        }
    }
}

//after all the data has been copied into the
↪  buffer, figure out where the final FFD9 was and
↪  start writing data there.

size_of_result=( (end_marker+1) + 3 + chunk_size) *
↪  sizeof(unsigned char); // size of

result = <bytes of data from jpeg> + <3 bytes for
↪  our trailer> + <size of data to hide>
```

Because there is no guarantee that our encrypted chunk will not contain the value 0xFFD9, we added 3 additional bytes of data after the final 0xFFD9, to create a 5 byte signature of 0xFFD924FFD9 at the end of our JPEG data, reducing the odds of the specific series from randomly occurring in our data from 1 in 65,536 to less than 1 in 1,099,511,628,000.

```
//write our custom trailer in so the extractor knows
↪  where the data begins
data_to_write[end_marker+1]=0x24;
data_to_write[end_marker+2]=0xFF;
data_to_write[end_marker+3]=0xD9;
```

After our custom trailer has been added, the contents of the chunk are appended to the data already in the buffer, and then the original file is overwritten with the data contained in our buffer.

D. Extract Function

At reassembly time, our encrypted chunks are recovered from the JPEG files in a manner similar to how it was initially hidden. Our extraction function has two parameters: a string containing a path to the file to extract data from, and a pointer to an int which is declared in the reassembly function. Because the size of the payload in bytes is frequently not evenly divisible by the number of JPEGS in the DCIM folder, it is likely that one JPEG file will contain a chunk which is smaller than the chunks hidden in the other files. To resolve this, the reassembly function has an int variable that represents the size of the chunk being returned by the extract function, and it exposes this variable to the Extract Function for modification by passing it this pointer.

The extraction function copies the specified JPEG file into a buffer, and then iterates over the data until it locates the five-byte signature we set in the hiding function. Once this signature is encountered, the size of the chunk to be extracted is calculated by subtracting the number of bytes read at that point from the size of the file, and is stored by dereferencing the int pointer passed to the function. A pointer to the first byte of the extracted chunk is then returned by the extraction function.

```
int j = 0;
int end_of_jpg= 0; // find the jpeg ending

fread(jpg, 1, len, jpgf);
for(j; j < len; j++)
{
    if(j > 3)
    {
        if(jpg[j-4] == 0xff &&jpg[j-3] == 0xd9
        ↪  &&jpg[j-2] == 0x24 && jpg[j-1] == 0xFF
        ↪  && jpg[j] == 0xD9) // found the jpeg
        ↪  trailer
        {
            end_of_jpg=j;
        }
    }
}

size_t size_of_chunk = size_of_result -
↪  (end_of_jpg+1);

unsigned char *chunk= (unsigned char*)
↪  malloc(size_of_chunk); // make space to copy
↪  chunk into

for(int i=0; i< size_of_chunk; i++)
{
    chunk[i] = jpg[end_of_jpg+1+i];
}

free(jpg);
fclose(jpgf);

*size = ((int) size_of_chunk);

reurn chunk;
```

E. Reassemble Function

The logic for reassemble will be almost same with split. We first calculate the number of jpeg in the directory. However, since we do not have the size of data chunk or size of the file, we will call the extract chunk function to get the size of each data chunk. Then we will allocate an array to store the data extract from the picture. The array's size will be calculate by multiply size of chunk with the number of pictures in directory. As mention above in split, this array will be equal or larger then it need to be because there will be picture without any hiding data in the directory. We will also have a variable to keep track of the real size.

```
while (((entry = readdir(dir_path)) != NULL) &&
↪  (!done))
{
    if (strcmp(entry->d_name, ".") != 0 &&
    ↪  strcmp(entry->d_name, "..") != 0)
    {
        // create new full path
        strcpy(full_path, jpg_directory);
        strcat(full_path, "/");
        strcat(full_path, entry->d_name);

        // if there are still byte to read from
        ↪  encryption file:
        //printf("%s\n", full_path);

        data = extract_chunk(full_path, size);

        //printf("%d", size_of_chunk);

        if (full_size == 0)
        {
           file=(unsigned char*)malloc
              (size_of_chunk*file_count);
        }

                // If there are data hidden in the
                ↪  picture
        if (size_of_chunk != 0)
        {
            for (i = 0; i < size_of_chunk; i++)
            {
                file[j] = data[i];
                j++;
            }

            full_size+= size_of_chunk;
        }
        else
            done = 1;
    }
}
```

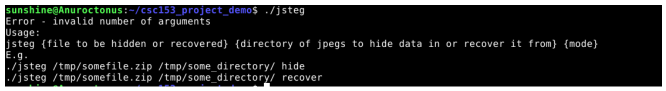
After got all the data reassemble, we will store it back into a file, then decrypt that file and we will achieve the original data.

V. Usage

Users can compile the program with command:

```
$ gcc jsteg.c -o jsteg -lcrypt
```

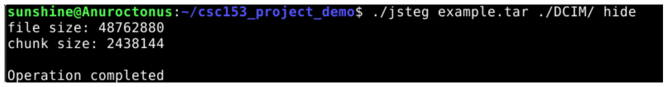
Then to run the application, users need to provide 3 arguments. The first argument will be the path to file or data that users want to hide or recover. The second argument is the path to image directory. And the last argument is the mode (hide or extract). If any of these three arguments are missing or if there are more arguments than the program need, a error message will be printed out to the screen, with instruction to compile the program.



```
sunshine@Anuroctonus:~/csc153_project_demo$ ./jsteg
Error - invalid number of arguments
Usage:
jsteg {file to be hidden or recovered} {directory of jpegs to hide data in or recover it from} {mode}
E.g.
./jsteg /tmp/somefile.zip /tmp/some_directory/ hide
./jsteg /tmp/somefile.zip /tmp/some_directory/ recover
```

Fig. 1. *Usage message*

Below is an example run when using the application to hide a zip file data. The program will print out total size of hidden file, and the size of each chunk.

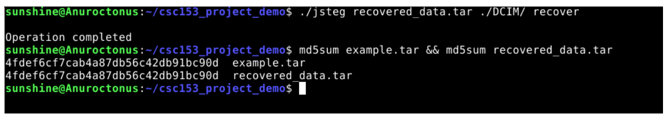


```
sunshine@Anuroctonus:~/csc153_project_demo$ ./jsteg example.tar ./DCIM/ hide
file size: 48762880
chunk size: 2438144

Operation completed
```

Fig. 2. *Hide the file into pictures in DCIM directory*

After reassemble the data, we use MD5 checksum to verify the integrity of our program. As can be seen below, the check sum value of original file and reassemble file are identical, which mean the process succeed.



```
sunshine@Anuroctonus:~/csc153_project_demo$ ./jsteg recovered_data.tar ./DCIM/ recover

Operation completed
sunshine@Anuroctonus:~/csc153_project_demo$ md5sum example.tar && md5sum recovered_data.tar
4fdef6cf7cab4a87db56c42db91bc90d  example.tar
4fdef6cf7cab4a87db56c42db91bc90d  recovered_data.tar
sunshine@Anuroctonus:~/csc153_project_demo$ 
```

Fig. 3. *Recover the hidden data and check integrity using MD5 checksum*

Here are two pictures before and after hide the data. There are no different between them just by looking at the pictures.

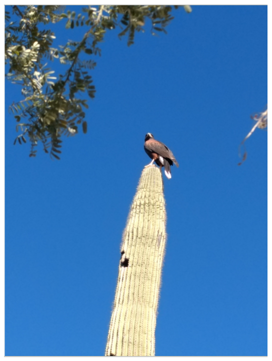


Fig. 4. *Picture before hidding secret file*

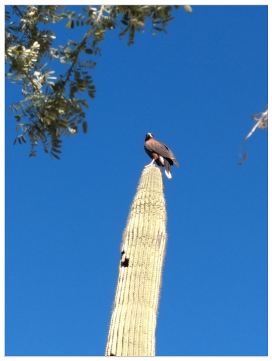


Fig. 5. *Picture after hide the secret data*

VI. Result

Our application, which we ended up referring to as jsteg, successfully provides the functionality which we desired. While our strategy favors capacity over all, we feel this strategy strikes a desirable balance between robustness, capacity, and perceptibility for the context in which it is intended to be used. The main advantage our strategy offers is the ability to hide a significant amount of data in each file.

The other strategies we examined (LSB hiding and utilizing file system slack space) both had the drawback of severely limiting the amount of data that could be hidden per file. The amount of slack space available would vary with the size of the file, and potentially cluster size, as either FAT32 or exFAT filesystems might be encountered on cards used in different cameras, but generally would be measured in kb per file. Least significant bit hiding would increase the capacity, but still limits us to payloads of around 1/8th to 1/4th of the original file's. Increasing the potential size of the payload by using more significant bits has the disadvantage of degrading the cover image.

While LSB hiding strategies have some advantage over ours if the file were to be suspected of concealing data (the presence of non-image data is significantly easier to detect in ours), the practical benefit of that advantage seemed very limited in our scenario, as our primary goal is to avoid having the files scrutinized closely in the first place, and LSB hiding is a very well known method of concealing data. In the case that one of our image files is scrutinized closely, the concealed data is both encrypted and broken up between the files which should present a significant challenge to an attacker trying to recover the data. One minor improvement which we did not implement would have been to append a final 0xFFD9 after the chunk, which would make the file look less conspicuous if examined in a hex editor.

In terms of perceptibility and robustness, our implementation meets what we feel are very practical goals. As the resolution of digital cameras has improved, the size of image files contained on storage cards has increased dramatically. This means that JPEG files of sizes of up to 5 and 10 Mb would be unlikely to arouse suspicion if they are high resolution photos. Further, it would seem quite improbable that most persons, including fairly knowledgeable computer users, would be able to determine the appropriate size for a high resolution JPEG file simply by looking at to a degree of accuracy that would arouse any suspicion, so long as some common sense is employed (e.g. the size of the JPEGs is not increased to such a wild degree as to be obvious). As the stego-files are intended to be transported on a flash memory card within a digital camera, they are unlikely to be subjected to any type of intentional modification.

VII. Work Distribution

To implement our plan Dean was in charge of researching and creating a feasible way to conceal and retrieve chunks of the data file that needs to be hidden. Tran was in charge of researching and putting together the split function that will divide the file into chunks and reassembling them afterwards. Initially, Jessica was in charge of researching and installing a library that will encrypt and decrypt data. At first we considered incorporating a premade library encryption and decryption program to help us with our project, but we ran into some problems when installing the libcrypt program onto the computer. Since Tran had previously studied cryptography, we end up implemented file encryption/decryption functions based on an algorithm she had learned previously.

VIII. Conclusion

Yet, the technique we use to hide the data is unsophisticated compare to other strategy such as LSB. Besides, the program still has some constraint, and maybe easy for steganography analysis program to detect it. However, regardless, jsteg is a really nice simple tool, and building it was an excellence learning opportunity. We enjoyed and had lots of fun implementing all parts of the project.

References

[1] R. Mishra and P. Bhanodiya, "A review on steganography and cryptography," 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, 2015, pp. 119-122, doi: 10.1109/ICACEA.2015.7164679.

[2] "What is Steganography?," retrieved from https://www.clear.rice.edu/elec301/Projects01/steganosaurus/background.html; Accessed Nov, 2020.

[3] S. Thampi, "Information Hiding Techniques: A Tutorial Review," in ISTE-STTP on Network Security & Cryptography, LBSCE 2004 https://arxiv.org/ftp/arxiv/papers/0802/0802.3746.pdf,

[4] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," in NIST Special Publication 800-38D, Nov 2007

[5] LSoft Technologies Inc,“JPEG Signature Format,” retrieved from https://www.file-recovery.com/jpg-signature-format.htm; Accessed Nov 2020.