PhotoEncryptor: Channel Switching Encryption Engine

Marcos Barbieri

200-53-364

Professor Rivas

MSCS-630L

Marcos Barbieri
Professor Rivas
MSCS-630L
Project Proposal

Abstract

The proposed project will be designed to encrypt messages within images using concepts presented in the Least Significant Bit (LSB). The app will have a similar design to a photo upload application. The focus of this project has shifted slightly since its proposal. As the project proceeded, further interest was found in encrypting messages within images. This is something that is not commonly used, and as a methodology is interesting to test. Therefore, the idea is to encode a message and place it within an image to provide obscurity.

Introduction

The intent of this methodology, is to apply some of the concepts in image steganography. This means that the sender of a message will encrypt a message within an image. Images are often disregarded as means of transferring information. Thus, this makes them great for the encoding of private information. In addition, it can be a form of layered security. This means that after encrypting a message, said encrypted message can also be encoded into the image, creating more work for an attacker with intent of decrypting any messages being transmitted.

Background

Terminology

It is important to understand some terminology before proceeding with the argument. Each term defined has a definition specific to this project, but may or may not be the accepted use of the term in other projects. That being said

Plaintext refers to the original message that the user wrote down

Cover Image refers to the image that will contain the hidden data

General Information

The native iOS programming language, Swift, uses an object called a `UIImage` to represent image files on an application. Through the `UIImage` object we can obtain pointers to the image in memory and obtain the bytes that make up said image. These bytes represent the RGBA codes that make up the pixel. The acronym RGBA represents the color space and stands for Red Green Blue Alpha, in which each corresponding code corresponds to each word in the acronym. Thus, there is a byte value for the amount of red, green and blue in the pixel as well as a byte for the alpha channel information.

Methodology

Channel Space

This algorithm applies to the channel space, which can be regarded as each byte in the pixel hex color code. Suppose we have a pixel with the channel code

#01AAFF 55

As indicated the first byte of the code contains the pixel value for the red component, clearly marked in red. The second byte shows the green pixel value, and the third shows the same for blue. Now, the last byte represents the alpha component, deals with transparency and depth of images. Each of these components is called a *channel* and contributes to the visual aspect of each pixel.

Since we have 4 bytes in each pixel, we can replace any of these with ASCII character data (Unicode will have a different process). Thus, when receiving the plaintext input, each character must be converted to its ASCII character value, and should replace a channel byte with the character byte.

Channel Switching

When replacing channels with character data, if only one channel is replaced for the entire image then it can be easy for an attacker to brute-force and check every channel for hidden messages. Thus, this is where the idea of channel switching can be introduced.

In order increase diffusion, there must be a "channel switching" operation that needs to be made. However, this will require a key to be constructed. To follow the approach of AES, round keys can be created in the same way they are created in AES. The constructed key can identify the channel that the data is actually encoded on. Therefore, a simple modulus operation can calculate the channel. Consider the channel space $\boldsymbol{G}$, it is clear that the data must be encoded in one of the channels within the channel space, denoted by $\boldsymbol{G_i}$. $\boldsymbol{G_i}$ can be calculated by taking one byte of the constructed key, denoted by $\boldsymbol{K_i}$, and performing $\boldsymbol{K_i}$ % $\boldsymbol{n}$ where $\boldsymbol{n}$ is the number of elements in set $\boldsymbol{G}$. Thus, each pixel within the cover image will have embedded data at different locations in each byte.

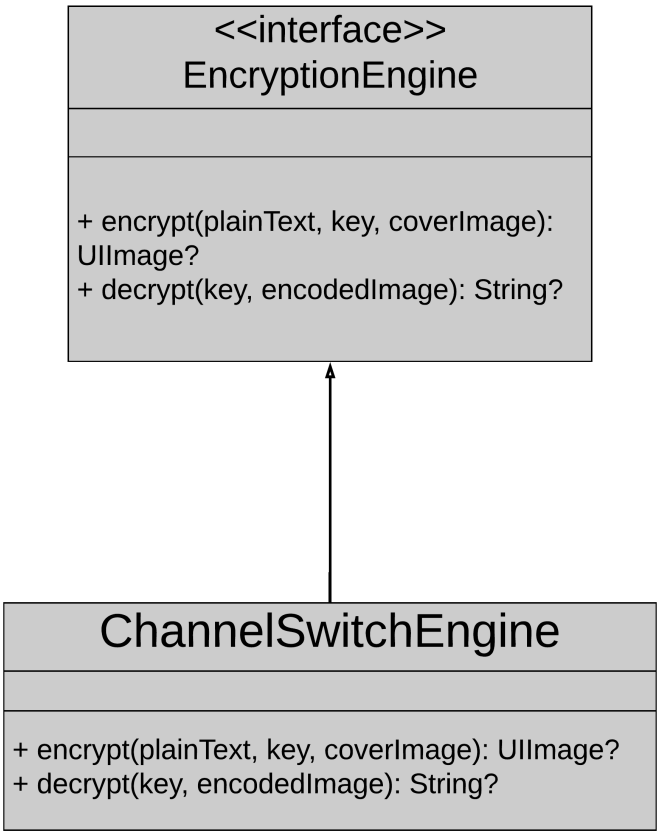
As of now the only limitation of the key is that it must be the same length as the plaintext. However, a user may enter any key they like and the program can provide the padding to create the appropriate length key.

Implementation

Class Hierarchy

The implementation of this algorithm can vary in approach, but only the Object-Oriented approach will be shown. Below is the class hierarchy for the current application

On



Since this implementation the `EncryptionEngine` is actually a `Protocol`, but would be more commonly recognized as an `Interface`. We see that this `Protocol` requires the implementing class to create an `encrypt(plaintext, key, image)` and a `decrypt(key, image)` method. This makes it simple to inherit and allows the implementing class to be lenient in the way it handles the encryption.

The advantage of this approach is that it allows for other developers to implement multiple encryption engines and perform multiple levels of encryption on a plaintext and cover image. This means that in addition to the steganography, developers may choose to run the plaintext through an AES encryption engine before hiding the data in an image.

Experiments

All experiments performed were run on the iOS 8 simulator on macOS *High Sierra*. The simulator contains about 5 stock images that were used for testing. When testing the first image, a plaintext of `hello` with a key of `hello` was successfully encrypted in the image. Since the message was small there was no apparent change to the pixel quality of the image. Thus, to ensure that the image was in fact different than the original each image was placed in an MD5 Checksum generator. The plain image resulted in a checksum of

`2E838298882840700F92D77B15DCC1F`

while the encrypted image resulted in a checksum of

`FEE0DA13348AC8DE4A3E6FE231991C`

showing that the encrypted image had in fact changed. This experiment was carried out for all images in the iOS Simulator stock image set, and proved to be successful.

This, however, came with a large shortcoming. Every decryption run would display the incorrect message that was provided with the key and image. Thus, this proved to fail for all test cases.

Analysis

Upon further research, it was soon discovered that the incorrect decryption had to do with the compression of the image. As of now iPhones support multiple file types for images, but the default is the file format known as JPEG. The JPEG format uses lossy image compression to conserve space in the store, thus the image was erasing the pixel data that was written to the image after compression, and the checksum would register the image with a different signature because it had been changed after compression. This comes about with an easy, but inconvenient fix, which is to only use PNG files for encryption. This is because PNG file formats use lossless compression, which will retain any information written to the pixel data after compression.

Conclusion

Ultimately, the fix for the image compression was relatively simple, in that the only requirement was to save the new image that is created as a PNG file. Swift provides an implementation of a method that does this automatically, but for any implementations in other languages, the developer must be sure to write a library (or to use a 3rd party library) for this very purpose. After the PNG constraint was added to the application, the message was decrypted properly and the bug proved to be resolved.

As further comments to this project, there are things that were not implemented in this project that would add diffusion to the encrypted data and can make it harder to reveal the plaintext. Things like using the full scope of the least significant bit algorithm, by analyzing what colors are least used in the image, and only replacing those channels (and switching every so often), using AES to encrypt the plaintext, as well as hiding the data with pixel padding in the image (not every pixel, but perhaps every other, or third).

Nevertheless, as a proof of concept, this can be used to further implement applications that hide data in images in different ways, and proves to be confusing to any attacker.

References

Savani, Hussain. "Image Steganography." *LinkedIn SlideShare*, 2 Jan. 2014, www.slideshare.net/hussainsavani/image-steganography.

Pandey, Aishwarya, and Jharna Chopra. "Steganography Using AES and LSB Techniques." *International Journal of Scientific Research Engineering & Technology (IJSRET)*, vol. 6, no. 6, June 2017, pp. 620–623.