CaracalSteg: A Dart JPEG Steganography Library and Companion App

A Technical Report
presented to the faculty of the
School of Engineering and Applied Science
University of Virginia

by

Dylan Cao

April 8, 2021

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Dylan Cao

Technical advisor: David Wu, Department of Computer Science

Introduction

Digital steganography is the process of embedding hidden data inside another file, such as an image, movie, or audio file. It is not a substitute for encryption. The use case for steganography is to reduce (but not eliminate) the chance that a message's existence is detected by third-parties. Unless used alongside encryption, someone who has detected the message is usually able to decode the data if they know the steganographic scheme. While it is thus less secure than encryption, it is more subtle: it is harder to tell whether a cover file contains steganographic data than it is to determine whether a file is encrypted, as encrypted files are often simply unreadable.

Image files are well-suited to be cover files, since most online social media and messaging platforms allow for users to share image files with each other. Some users of these platforms may wish to share short messages with each other that are not obviously visible to the platform owners or to the general public, which image steganography can achieve. However, most image sharing services will use lossy compression on images to save storage costs; this includes techniques such as converting images to JPEG files or recompressing JPEG files at lower quality values. Traditional steganographic techniques such as least-significant bit steganography will not survive the conversion or recompression process; our goal was to implement steganography that can survive these changes. The result was CaracalSteg, a Dart language library and command-line app capable of robustly embedding data in JPEG files that survive recompression along with a Flutter mobile application front-end.

Background

JPEG is a very common, lossy image format. It operates on 8x8 blocks of pixels using the Discrete Cosine Transform (DCT) (Al-Ani & Awad, 2013; Chang et al., n.d.). At a high level, the DCT used in JPEG compression decomposes an image into multiple frequency components. The low frequency components of an image are the most important for human perception; high frequency components generally make up subtle details which may not be immediately obvious to users. Therefore, JPEG operates by preserving low frequency data as much as possible, but compresses high frequency data using quantization to save space. While this is generally not immediately visible at lower compression levels, it *does* result in changes to the underlying pixels. The underlying data behind a JPEG image is RGB, with 8-bits per color; the less-significant bits of each pixel tend to be altered after JPEG compression. Some traditional methods of steganography use the least-significant bits of RGB data to store their hidden message; this is not suitable for JPEGs since that data is extremely likely to be lost.

Since JPEG preserves low frequency data well, though, we can exploit this by embedding our data into the low frequency components. While the DCT used by JPEG is tricky to implement, other simpler discrete wavelet transforms are able to decompose images into low and high frequency components in a sufficiently similar way. Specifically, we have implemented a slight modification of the Haar discrete wavelet transform described by Xu et al. (2004). Though the algorithm is pre-existing, we could not find an implementation of it suitable for public use. Other researchers found that some existing steganography programs may produce images suitable for upload to Facebook, which is similar to what our program aims to do (Hiney et al., 2015). However, two of the three suitable programs identified as working for this use case are command-line only, and the third, JP Hide & Seek, has a rather unfriendly interface (that does not even correctly render on our Windows 10 laptop—the image below was not cropped, the UI

is actually cut off). Thus, our project seeks not to be novel in terms of functionality, but in terms of usability.

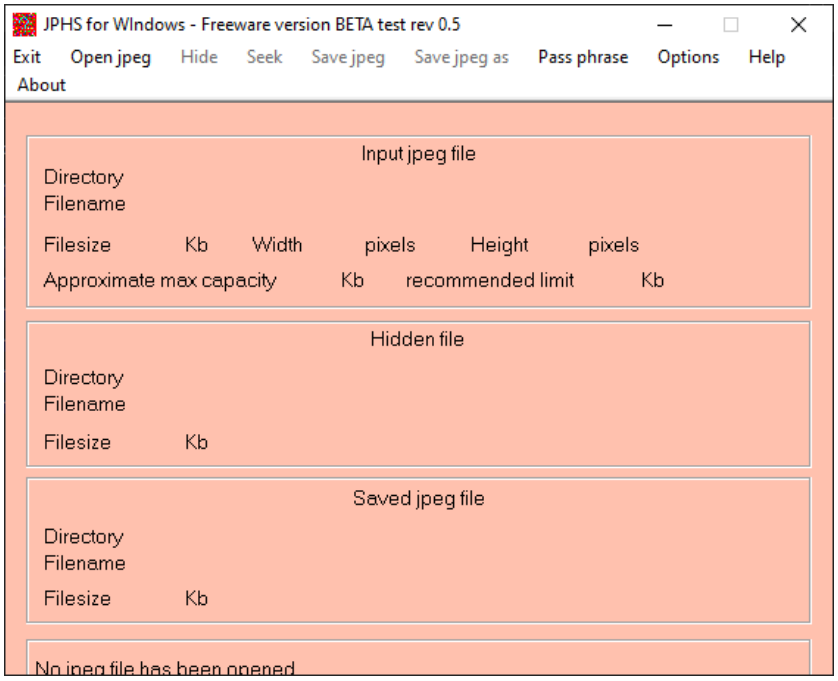

Figure 1 - JPHS screenshot, program downloaded from http://io.acad.athabascau.ca/~grizzlie/Comp607/jphs05.zip.

Implementing the Library and CLI

We chose to write our program in the Dart programming language (https://dart.dev). The reason Dart was chosen is that it is the language used by Google's cross-platform Flutter mobile application framework (https://flutter.dev), which would simplify front-end development later. The library and CLI implementation are available at https://github.com/Rayhawk11/caracal_steg. Binaries for the CLI for Windows x64 and Linux x64 are also available on that page.

Overall Architecture

The overall steganography process has multiple steps. First, we need the message to hide: we chose to limit ourselves to handling a subset of ASCII character data, specifically printable

characters between 32 and 126, inclusive, rather than handling arbitrary data for simplicity of implementation. The message then needs to be encoded into bits that can be inserted into a cover file. The cover file is then read in by the program, the encoded data is inserted, and an output JPEG file is produced.

Least-Significant Bit (LSB) Steganography

We started using an implementation of LSB steganography as a baseline for testing purposes. LSB is not a robust technique on its own, so this is primarily for experimentation. First, the message is encoded into a bitstring. Originally, our implementation just used the bits of the ASCII message; later on, we used a repeated 256-bit Hadamard code for redundancy. After the bitstring to embed is generated, it is then inserted into the file using the nth least-significant bit of an RGB value of a pixel. This is configurable, but defaults to the 4^{th} least-significant bit; for JPEGs, less significant values result in unusable levels of data loss even with little compression applied to the output image. Pixels in the image are used in a left-right and top-down order: the first pixel modified is in the upper-left corner of an image, then the pixel to the right of that, etc. The channels used are in RGB order. Given 3 bits of information to embed, the first bit goes in the nth LSB of the R channel, then the G channel, etc.

This method by itself fails to produce usable JPEG steganography unless a very high bit is used, such as the most-significant or second most-significant bit. This, however, defeats the point of steganography: the alterations to the image if using the high order bits of the pixels is extremely obvious. However, now that we had an LSB steganography implementation, we could work on experimenting with error-correcting codes.

Repetition Codes

Repetition codes are very simple error-correcting codes. Instead of embedding a message once, we simply embed it as many times as the file allows. The capacity in bits of a given image is (width * height * 3): three bits for every pixel in the image. We can thus divide by that by our message length to determine the number of repetitions we can use. For example, if our message is “AB” (two 8-bit characters) and we have a 32x32 image, we can repeat the message 32*32*3/16 = 192 times.

In our implementation, the message is repeated in whole. That is, if we have the capacity for two repetitions of our message “BAR,” it is embedded as “BARBAR.” It is also possible to repeat in a character-wise fashion, such as “BBAARR.” Repeating in whole is theoretically helpful for error-correcting capability in our application. With character-wise repetition, each character is embedded in a relatively local set of space in our image; if compression destroys that whole space, the character is lost even with repetition. With message-wise repetition, each character is embedded multiple times throughout the image, so local errors still allow us to recover the original character.

In decoding, we chose to assume the most-decoded character in each position was the correct one. As an example, if we decode the repetitions “AB”, “AC”, “BB”, and “CD” from a cover image, we output the string “AB.” “A” appeared twice in position 1 (compared to B and C, which occurred once). “B” appeared twice in position 2, which is more than “C” and “D” did. The other option is to use a bitwise majority. Since each character is an 8-bit string, we could count up the number of times the bit in position 0 for the first character is 1, then the bit in position 1 for the same character, etc.; if a bit in a position is 1 a majority of the time, we assume

1 is the correct value; otherwise, we would assume it is 0. Experimenting with this decoding scheme found that it is generally less accurate, so the first scheme was used.

To use this with LSB steganography, we simply use the repeated message converted to bits as the bitstring passed to the LSB embedding step. We found this did somewhat improve our accuracy, but improvements can be made by combining error-correcting codes.

Hadamard Error-Correcting Codes

The Hadamard error-correcting code is a more complex but still easy to implement error-correcting code (Malek, n.d.). Since each character is 8-bits long, the Hadamard code we implemented uses 256 bits to encode 8-bit words. The error-correcting capability of this code is [0, 64] and [192, 256] bit flips. If a number of bits outside those intervals is flipped out of the 256-bit code, then the decoded 8-bit word will be inaccurate. The code length of 256 is extremely high compared to our message length of 8; however, this is the trade-off for the code's ability to correct a very wide range of errors.

To encode an 8-bit message into a Hadamard code is quite simple. First, we need to generate the 256x256 Hadamard matrix. Matlab provides the hadamard() function to do this; we placed the result into hadamard_matrix.dart. We use the row of the matrix corresponding to the decimal value of our 8-bit word as the Hadamard code. That is, a message of 0b00000001 has a decimal value of 1 and corresponds to the second row of the Hadamard matrix (due to zero-indexing). The rows of the Hadamard matrix then need to be slightly modified to produce our binary error-correcting code: originally, it consists of -1s and 1s; the negative 1s are interpreted as 0s in our resulting binary code. This 256 length bitstring is what is passed on for embedding.

Decoding is slightly more complicated. Assume we have retrieved the 256 length Hadamard coded bitstring, *c*, from our cover file and wish to retrieve m¸ the original character. We convert *c* to a vector and multiply it by our Hadamard matrix, H. c*H yields *v*, a "magnitude vector." The index of the highest absolute value in *v* is *m*. For example, if *v* = [-5, 4, 2], then *m* = *0* since 5 is the highest absolute value in the magnitude vector.

Repeated Hadamard Error-Correcting Code

We can combine error-correcting codes for even more robustness. Instead of embedding our Hadamard-coded message once in a file, we can repeat it in the file. A message with two characters M_1 and M_2 is then converted into Hadamard codes C_1 and C_2. The codes are then embedded in the file as $C_1C_2C_1C_2C_1$… in the manner described in the repetition codes section. Each repetition is then decoded into a character, and the most-commonly occurring character in each message position is assumed to be the correct one. We have thus combined our Hadamard code with our repetition code.

Haar-Wavelet Transform Method of Embedding

Experiments showed yet more improvement in decoding accuracy, but still not enough for LSB steganography to be viable. We turned to Xu et. al's 2004 paper for a better method to embed our Hadamard-repeated codes in the cover image. The basics are described here, but the original paper should be referred to for detailed formulas if desired.

The Haar discrete wavelet transform (Haar DWT) decomposes a matrix of an image (or other values) into four matrices of frequency coefficients. The dimensions of each coefficient matrix are half that of the original image; for an 16x16 image, each coefficient matrix is 8x8.

The important set of coefficients is denoted *cA* and represents the low-frequency components of the image. We can use the Haar DWT to produce cA_2 by decomposing *cA*; then, we do the same to cA_2 to produce cA_3. The dimensions of cA_3 are width/8 x height/8 of the original image. Recall that the DCT used in JPEG compression operates on 8x8 blocks; this is not a coincidence. As Xu et. al explain, the values of cA_3 correspond highly to the low-frequency components most preserved by JPEG's DCT-based compression.

The integer Haar DWT described in the Xu et. al paper is implemented simply using basic matrix operations, and produces a cA_3 matrix that also consists of 8-bit integers. It is fully and losslessly invertible as well. The simplicity is a major advantage over attempting to implement a DCT transform, which would output floating point values and be fairly vulnerable to floating point rounding errors. Also, the fact that cA_3 consists of 8-bit integers is highly advantageous for our application: we can embed our coded messages into the last-significant bits of cA_3, then invert the Haar DWT to produce our output bitmap. Xu et. al suggests using the 3rd significant bit of cA_3, which we do by default (though our implementation is configurable). Otherwise, the major difference between our implementation and theirs is that they use a different error-correcting code prior to embedding.

In our implementation, when using the Haar DWT method, the order of embedding is left-right top-down RGB in that order. In other words, the value used to embed our data is the upper left value of the R channel's cA_3, then the top left value of the G channel's cA_3, etc.

Finalized High-Level Procedure

Our finalized high-level procedure is the following. First, we take in an input message and input image. We resize the input image if necessary, since the dimensions of the image we

work on need to be multiples of 8 for our 3-level Haar transform to work. We then calculate the number of repetitions possible given our input message. This follows the formula (width*height*3) / (message length * 256 * 64): width * height * 3 bits are available to us in total. The Hadamard code for each character is 256 bits long. Finally, our capacity is reduced by a factor of 64 since there is only one cA_3 value for each 8x8 pixel block. We then repeat the Hadamard coded characters as calculated above, and embed the final bitstring into cA_3. The Haar DWT is inverted to produce a bitmap (set of RGB pixels), and the bitmap is saved as a JPEG file of configurable quality (aka compression level).

Decoding the hidden message involves producing cA_3 from the input JPEG file, extracting the least-significant bits of cA_3, Hadamard decoding them, accounting for the repetition codes, and finally outputting the original message. The program must know the number of characters in the hidden message, or corruption will occur due to misaligned repetitions.

JPEG Quality

Users can provide a quality parameter between 0 and 100 to the program, corresponding directly to JPEG's quality parameter. This affects the size of the output image, the subjective quality of the output image, and the robustness of the message. Lower quality causes more compression, reducing the quality of the image and the survivability of the hidden message, but also reduces the file size of the image. The quality parameter defaults to 95, which has low amounts of JPEG compression artifacts and results in a very high likelihood that a message is fully decodable. However, our experiments found that messages tend to be decodable down to a quality of about 30, depending on the size in pixels of the image and the length of the message.

Command Line Interface

The CLI program is implemented in caracal_steg.dart. Documentation for how to use it can be found by running it with the --help flag. It supports both Haar DWT operations as well as the inferior LSB operations. Standalone binary executables for both Windows and Linux are available on the GitHub page. The CLI also implements a --resize flag for DWT encoding operations that is not simply a wrapper around the library API. If specified, the input image will be resized to the specified width before being operated on. The library API has not yet implemented this functionality, but rather expects users to have already performed this step if desired. The CLI also implements a --psnr flag to compute peak signal to noise ratio between the output image and the input image (a metric of how detectable the steganographic changes are) and --decodability (a helper flag to compute how decodable a message was after being embedded).

Testing the Haar Steganography Algorithm

As a result of time constraints, systematic testing and data collection could not be performed. However, we did perform some less formal testing and report our observations here. These results specifically pertain to the Haar algorithm; we will not report results on the less interesting LSB algorithm.

First, JPEG quality levels of roughly 30 or higher generally allow for a message to be fully or nearly fully decodable *before recompression* as long as the image is sufficiently large. By before recompression, we mean the output of the DWT encoding program, before being opened and resaved using a graphics program, uploaded to an image sharing service, etc. Messages are significantly less likely to become decodable if they are long enough to be near the

embedding capacity of the image unless image quality is high. This is since long messages being embedded in small images allows for fewer repetitions, reducing redundancy. This is significantly less of an issue at higher JPEG quality levels (90+) since Hadamard error-correction at those levels tends to be sufficient.

Second, at JPEG quality level 95 (henceforth "Q95"), our program seems to succeed (with some limitations) at its targeted use case: social media and online messaging. Some tests were performed in three forms of communications services: GroupMe, MMS (multimedia message service, used to text images over cellular data), and Facebook image sharing. At Q95, Facebook and MMS compress most images to around 60% of their original size (as outputted by our program). Despite this, moderately sized images (1920x1280 pixels, which is slightly more than a typical computer screen resolution of 1920x1080) are fully decodable even after recompression by the service. Subjectively, images with embedded messages do not appear obviously corrupted. However, differences can be seen when compared directly with the original cover image.

Figure 2 – Left shows sample image without message, right shows one with a Haar embedded message. Artifacting resembling odd squares and color shifting can be seen.

Limitations in message survivability primarily occur with larger images due to services' compression and resizing. If an image is to be shared over a lossless medium, for example Google Drive cloud upload, no issues occur with large images. However, 6000x4000 images (the resolution of an entry-level mirrorless camera) are extremely large and thus aggressively shrunk by MMS and GroupMe, including to the point of resizing them to smaller dimensions instead of just further compressing them. Resizing or cropping of an image with an embedded message is catastrophic and results in the message being more or less entirely lost. It is thus important when using this algorithm to downscale extremely large images to resolutions such as 1920x1080 or similar when sharing over lossy image services.

A Bug in the ml_linalg Dart Package

During the development of the steganography package, a bug was found in the ml_linalg package. As the name suggests, ml_linalg implements linear algebra operations which are essential to computing a Haar DWT. The bug was present in version 12.17.3 of the library, and a reproducible snippet was submitted to the library's author on October 14, 2020. The author confirmed the issue on October 15, and fixed it by October 22 in library version 12.17.4. The now-closed GitHub issue is at https://github.com/gyrdym/ml_linalg/issues/100.

In-Progress: Flutter Mobile App Front-End

We are developing a front-end for the steganography library using the Flutter mobile application framework. The code as well as an APK binary release for Android is available at https://github.com/Rayhawk11/caracal_steg_ui. At this time, the app supports Android only since we lack the hardware needed to build or test for iOS. However, the code base should be

able to run on iOS with little or no changes. Flutter also has desktop app support in beta, so in the future it is also possible that the app can be easily ported to desktop platforms.

The app is still in development and thus missing important features but is still usable. Right now, the app only supports the Haar DWT algorithm; in the future, a configuration setting can be added to switch to the LSB algorithm. The default should still be Haar DWT due to its superior robustness. Input validation is limited. A bug has been observed with currently unknown cause that results in the app failing to load upon being newly installed onto a device, though an app restart loads correctly. Future versions of the app will also need to have an image resize functionality implemented so that large images can be downscaled for faster processing and to avoid aggressive image resizing by online services. However, the core functionality works: the app can encode and decode messages embedded using the Haar DWT in a reasonable amount of time, and the UI is usable despite being simplistic and missing features. The app was tested on a Google Pixel 3 emulator, a Google Pixel 5, and a Google Pixel 2. The screenshots below are from the author's Google Pixel 5.

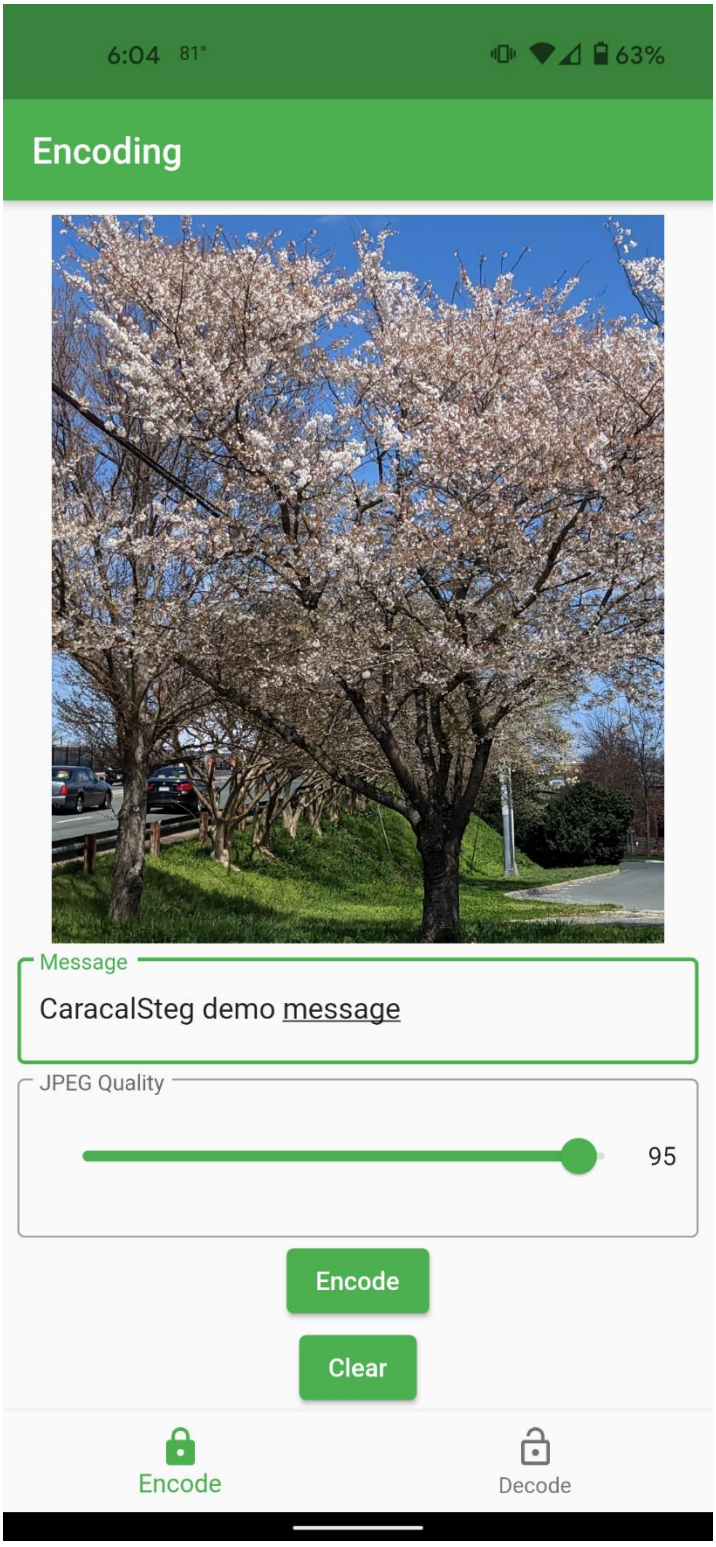

Figure 3 - Encoding screen

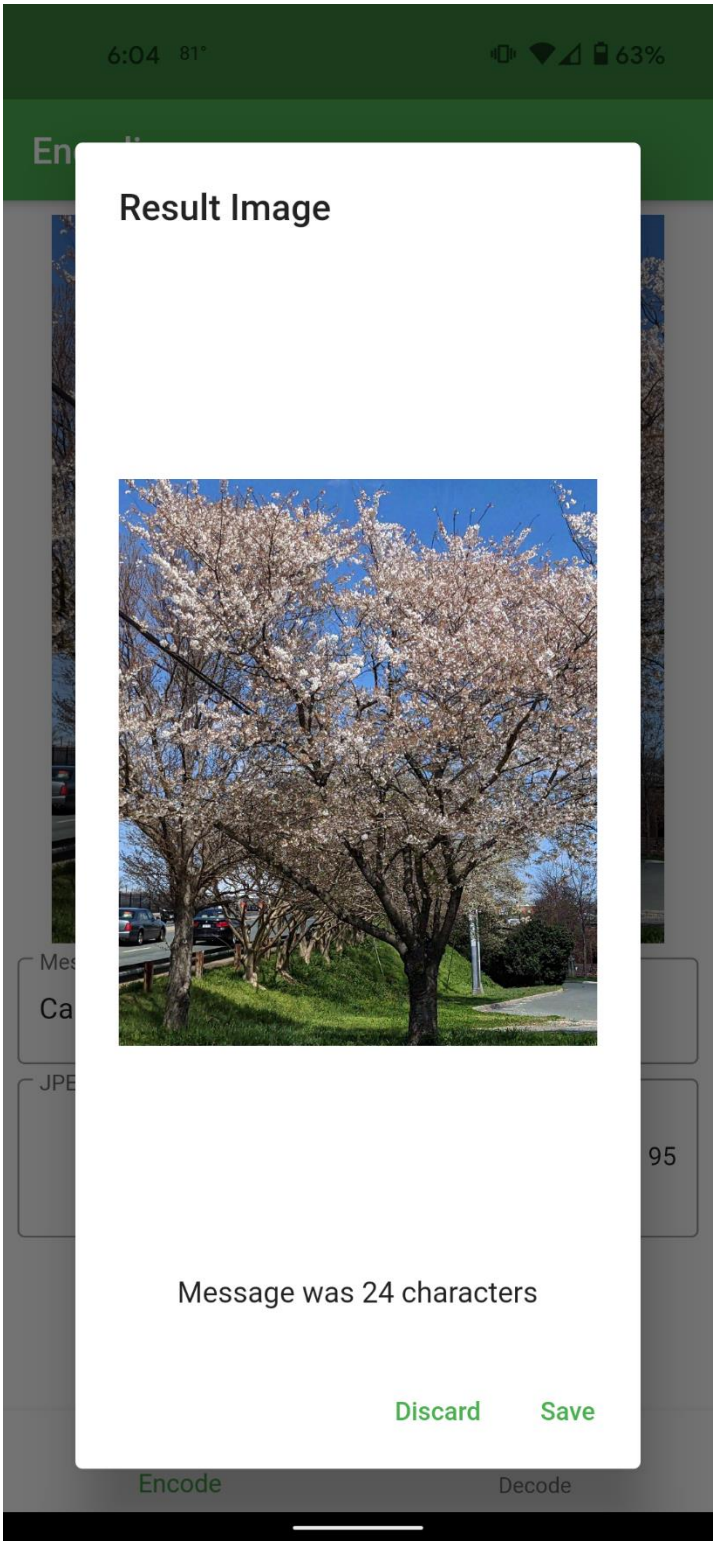

Figure 4 - Encoding screen result, with save or discord prompt

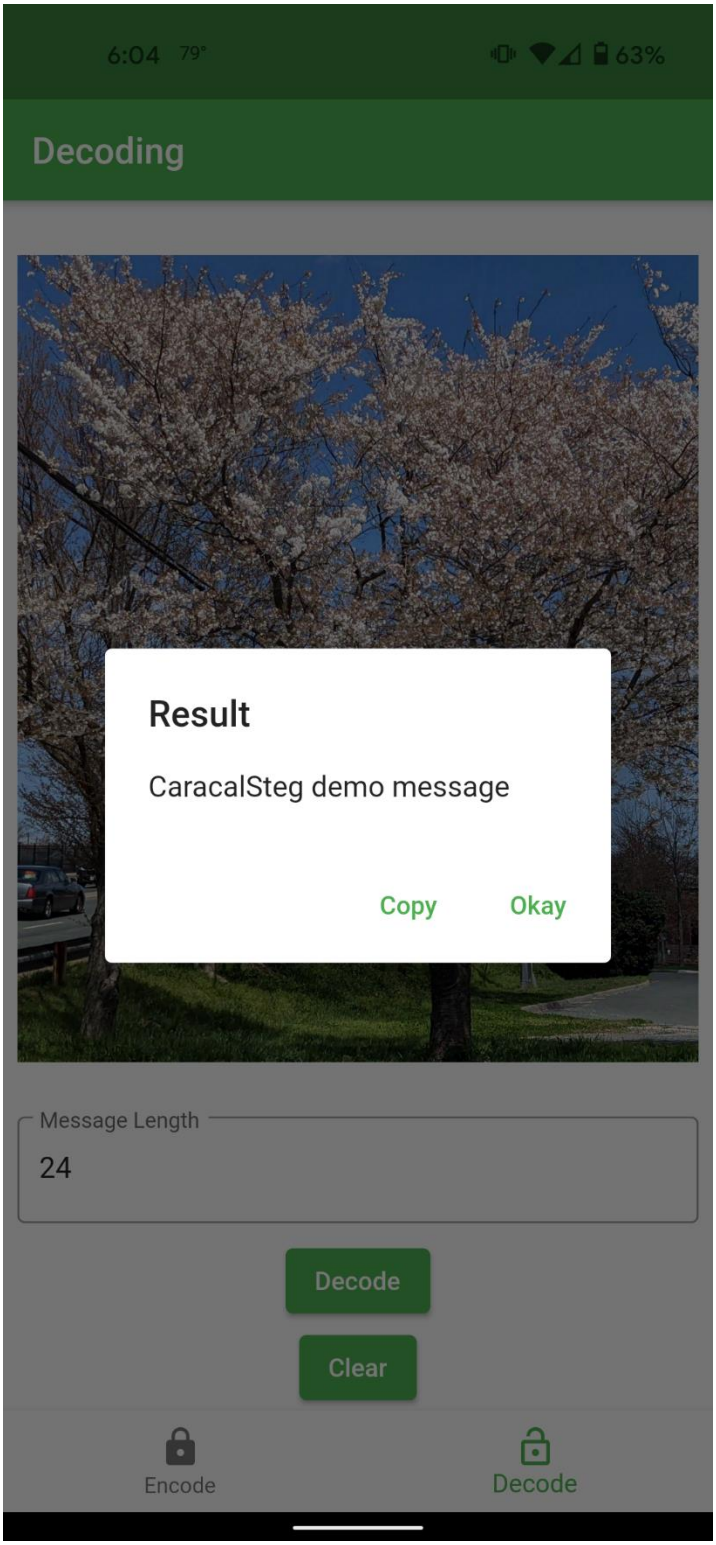

Figure 5 - Decoding screen showing with dialog box display the text embedded within the image

Future Work and Conclusions

There are a large number of ways to build upon this work. Systematic data collection about the robustness of the algorithm could be done across a wider variety of sample images, image sizes, and image sharing services. This data collection could also include measures of detectability, which was not focused on here since our main focus was robustness over security.

Basic schemes of encrypting the message could be introduced into either the library itself or the front-end. This would allow for the message to be secured even if it is detected and the embedding scheme is known by someone other than the recipient. This is still an advantage over simply sending encrypted messages without a cover image since it is more subtle.

Performance improvements could be made to the embedding and Haar transform algorithms. Major efforts were made during development to do this, but embedding moderately sized images still takes on the order of 30 seconds to a minute. Some lighter improvement may be possible without a rewrite, but greater gains might be obtained from porting the code to a platform native language like C or C++, since the Dart language lacks normal multithreading constructs and is not designed with performance as the highest priority.

Work could be done to ensure the library code is fully compliant with Dart library best practices. During initial implementation, this was a lesser priority; however, to be suitable for publishing to Dart's public package repository, some files should be renamed, certain internal-only functions should be made inaccessible to users, and others things can be done to improve code quality overall.

As previously mentioned, the UI also could use some work. The existing startup hang should be fixed. A progress indicator should be added, Haar DWT vs. LSB should be configurable, resizing functionality should be added, and there should be a way to cancel an

embedding/decoding operation. An effort could also be made to ensure the code works on other Flutter platforms like iOS or desktop.

Overall, though, the core functionality exists and seems to work as intended since hidden messages can be shared over certain image sharing services. However, the library and app are not completely finished and efforts will be made to improve and polish both the library and the front-end app.

References

Al-Ani, M., & Awad, F. (2013). The JPEG image compression algorithm. *International Journal of Advances in Engineering & Technology*, *6*, 1055–1062.

Chang, E., Fernando, U., & Hu, J. (n.d.). *Lossy data compression: JPEG*. Data Compression. Retrieved April 6, 2021, from https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/dct.htm

Hiney, J., Dakve, T., Szczypiorski, K., & Gaj, K. (2015). Using Facebook for image steganography. *ArXiv:1506.02071 [Cs]*. http://arxiv.org/abs/1506.02071

Malek, M. (n.d.). *Hadamard Codes*. https://web.archive.org/web/20200109044013/https://www.mcs.csueastbay.edu/~malek/TeX/Hadamard.pdf. Retrieved January 9, 2020, from https://www.mcs.csueastbay.edu/~malek/TeX/Hadamard.pdf

Xu, J., Sung, A. H., Shi, P., & Liu, Q. (2004). JPEG compression immune steganography using wavelet transform. *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, *2*, 704-708 Vol.2. https://doi.org/10.1109/ITCC.2004.1286737