Forensic analysis of steganography apps on Android

Wenhao Chen, Yangxiao Wang, Yong Guan, Jennifer Newman,
Li Lin, Stephanie Reinders

January 22, 2018

Abstract

The processing power of smartphones has promoted mobile applications of algorithms that were previously considered too computationally intensive, such as steganography. Steganography apps allow covert communication on a mobile phone using digital photographs on the phone. Digital forensic image analysis of photographs and other files aims to detect such covert activity. In this research, we ask the questions: how effectively can a steganography app be reverse engineered? How can this knowledge help improve detection of stego images and other steg-related files? Two Android steganography apps, PixelKnot and Da Vinci Secret Image, are analyzed. We find they are constructed in very different ways, and provide different levels of security for hiding a message. We also present results of performing detection on steganography files, that include images generated from the apps, with three different steg detection software packages. We conclude there is a need for further work in both reverse engineering of steganography apps and the ability to detect images produced by these apps.

Index Terms

Digital Image Forensics, Steganography, Steganalysis, Android Apps

I. Introduction

The field of covert communications has a long history. Encryption of a message, called *cryptography*, is a well-known method to communicate secretly, although by encrypting and sending, it is known that a message is being transmitted. *Steganography*, on the other hand, attempts to send a message without indication that it is being transmitted, to avoid detection of secret communication. This is sometimes called "hiding in plain sight." The word steganography originates from the Greek, meaning "covered writing." The first written evidence of steganography dates to the Greeks [1], where Herodotus describes how Histiaeus sent his slave to the Ionian city of Miletus with a hidden message to advise a revolt against the Persian king. The slave's head was shaved, a message tattooed on the scalp, and the hair grown back. Once the hair concealed the message, the slave was sent to the city's regent, Aristagoras. After shaving the hair, Aristagoras received the intended message. More recently, invisible ink and microdots have performed the same task. Digital versions of hiding messages are now available in code for computers as well as in apps for smartphones.

Digital steganography must hide a *message* or *payload* in the form of bits in a cover medium, also represented by bits, in such a way as to not arouse suspicion of its hidden content. The cover file is combined in some fashion with the payload and produces a *stego* file. The stego file is then transmitted to the intended recipient, who extracts the hidden payload. Sometimes a key is involved. The question remains, how can you change the bits of the cover medium, such as a digital photograph, a PDF document, digital audio file, or video file, to represent the payload bits, and then have the recipient successfully extract the payload once the stego version is received? Hiding a payload in a digital photograph can be accomplished by appending the payload after the End Of File (EOF) marker in jpeg images [2], [3], [4], in the color palette of a GIF image [5], or in the EXIF header of an image [6], [7]; in a PDF file [8]; or in the lower bits of a non-compressed RGB or grayscale image [9], or in the *quantized Discrete Cosine Transform* coefficients of a compressed jpeg image [10]. Digital audio [11] and video [12] files have also been used to hide payloads, as well as TCP/IP packets [13].

Detection of stego-related files is called *steg detection*. The existence of stego executables and related files on a system can indicate that steganography was perhaps performed, so steg detection can include the detection of such ancillary files that do not directly contain a payload. Patterns in the stego files such as an embedded *signature* or statistical properties can also be

exploited to perform steganalysis. Signature-based detection of a stego or ancillary file, based on specific characters or perhaps locations of specific added information, requires identification of a signature and creation of computer code to open the file and search for such signatures. Statistics-based detection considers statistical measures of a suspected stego file and searches for abnormalities that indicate a stego. Another type of steg detection involves the use of machine learning algorithms, and does not depend on signatures written explicitly into the stego file. This latter type of steg detection is termed *steganalysis* and is used and developed mainly in academic settings at the current time.

Although a mobile phone app can make steganography particularly easy to use, detection of stego images produced by mobile steganography apps has not yet appeared in the literature, to the best of our knowledge. Readily available for iOS and Android phones, mobile apps can conceal a text payload inside a selected photograph either residing on the phone, or acquired using the camera. With some apps, another file, such as a different image file, may be hidden. Although there are roughly 30 such steganography apps available, not all of them are stable, and may crash when using particular covering photos or large payloads, or on a particular model of mobile phone. The code used for the embedding part of the hiding process, where the bits of the cover image are changed to represent the payload bits, varies depending on the particular author of the app. "Bit embedding" refers to the process of changing the cover image bits to represent the message bits. In addition, not all apps have open-source code, which makes it difficult to reverse-engineer the code. Thus, signature-based investigation of these steg apps may not be easily performed, or even possible, if a signature does not exist. A machine learning classifier may provide more reliable steg detection than signature-based, assuming we can acquire enough image data to perform detection in this manner.

In principal, applying machine learning to detect stego images from a mobile phone camera is no different from the classical academic setting. By the "classical academic setting" we mean performing steganography or steganalysis using a known set of image data, and where the embedding algorithm is completely known and machine learning detection algorithms can be run. The object of many academic steganography algorithms is to demonstrate how difficult it is to detect a new embedding algorithm, using known "best detection" algorithms (typically machine learning algorithms). In the case of steganalysis, the goal is to show the new algorithm has clear performance advantages over existing ones. We do not present any steganalysis results using machine learning applications here, but include a short description for completeness. Instead

we focus on existing software that is available for practitioners used to detect the existence of steganography.

In this paper, we present our results from reverse engineering two Android apps, PixelKnot [14] and Da Vinci Secret Image [15]. Using our findings, we create a procedure to generate a large quantity of stego images on a computer using PixelKnot code without resorting to a human entering the information by hand, onto a phone, using the app. These two results have not been published in the literature, to our knowledge. We then use several software packages to perform detection of steganography, including testing of the stego images produced by the PixelKnot app, providing the first such evaluation of publicly available steg detection software.

The remainder of the paper is organized as follows: we discuss other work related to steg detection in Section II; in Section III, we present the two Android steg apps and a description of the reverse engineering process we created for this purpose; and in Section IV, we give a detailed analysis of the reverse engineering of PixelKnot and Da Vinci Secret Image apps. Finally, in Section V, we present our evaluation of three existing software programs that perform steg detection, and conclude in Section VI.

II. Related Work

In this section, we discuss existing methods for detecting steganography payloads hidden in image files. We also discuss existing tools used for reverse engineering Android applications.

A. Existing Steg Detection Approaches

We use the term *steg detection* to mean one of two conditions: (a) Discrimination between an innocent image and a stego image; or (b) Identification of a file that can be associated with a steganographic process. While steganalysis is commonly used to label the process described in (a), item (b) can be used to describe the identification of executable files, or other non-image files associated with producing a stego image. Forensic practitioners are interested in performing both (a) and (b). There are three basic types of steg detection:

1) Signature-based;
2) Hash-based;
3) Classification based on using features in a machine learning environment.

Next, we briefly describe the three different steg detection approaches.

Signature-based steg detection first identifies possible signatures that a steganography program writes to an output stego image, and then detects stego files using the signatures. Such signatures exist in different forms and types. As an example, a program could embed the same fixed bit string each time along with the payload; or, the embedding path could visit the same pixel locations in the same order regardless of the payload content, leaving a repeated pattern in stego files. Commercial tool Stego Hunt [16] and academic tool StegDetect by DC3 each provide signature-based steg detection. Given that the signature of the stego program is known, signature-based approaches can accurately identify stego images and possibly extract the hidden payload. However, this type of approach requires updating the detection code on a regular basis: any change in the signature can produce a different signature from the previous one, thus missing the detection of a stego file.

Hash-based steg detection involves the identification of a previous, identical stego file, such as an image identified in child pornography. Here, the exact same stego image hiding the payload is acquired by others, and so all the stego images are identical in a bit-by-bit comparison which results in identical hashes. This allows us to compare the hashed values of unknown images with a list of hashed values of known stego images. The hash values are stored in a database, and any new images needing to be analyzed have their hash value compared with others in the list.

Machine learning-based steg detection is a more complex detection approach. A machine learning classifier can be constructed (theoretically) to identify an unknown stego image, if training data is available, along with other caveats. We leave out the details, as it is beyond the scope of this paper. Two such classifiers are discussed in [17] and [9]. In order to use such classifiers, the steganalyzer must have access to a large amount of training data: typically, 700-6000 cover images, the same number of corresponding stego images, and a representative feature set and classifier. With these items and enough computing power, in many cases, it is possible to create a successful machine classifier to detect stego images.

B. Reverse Engineering Android Applications

In order to perform steg detection on stego images produced by an Android app, our approach is to inspect the Android program code that generates a stego image. By understanding how it works, it may be possible to exploit particular characteristics of how the code processes an image. We use reverse engineering techniques on two Android stego apps.

Reverse engineering is a commonly used program analysis technique. Analyzing an Android application often requires a reverse engineering tool to convert the application binaries (the APK) into a readable format. Android applications are developed in Java programming language, and compiled into *Dalvik bytecode* [18] that is similar to the Java bytecode. The *Dalvik bytecode* is then encoded and written into a DEX file within the APK. There are several existing tools capable of extracting and decoding the DEX file from the APK, and recreating the application code in the source or intermediate code format.

Apktool [19] is a tool for reverse engineering Android APK files. It can decode the DEX file into an intermediate code format called *Smali* [20]. *Apktool* can also decode the resource XML files including the graphical interface definition and the manifest file. Although *Apktool* does not translate the DEX into Java code, it provides an accurate representation of the binaries by avoiding loss in translation. *Dex2jar* [21] is a tool that can convert DEX to another intermediate code format, Java bytecode, by mapping the DEX instructions to Java bytecode instructions. *Java Decompiler* [22] can be used to decompile Java bytecode into Java source code, with possible loss of metadata and certain irreversible DEX code blocks. Using *Dex2jar* and *Java Decompiler* in combination, it is possible to recreate the Java source code from an APK file. However, due to the inconsistency of *Java Decompiler*, the resulting source code can only be used as a reference for the application analysis.

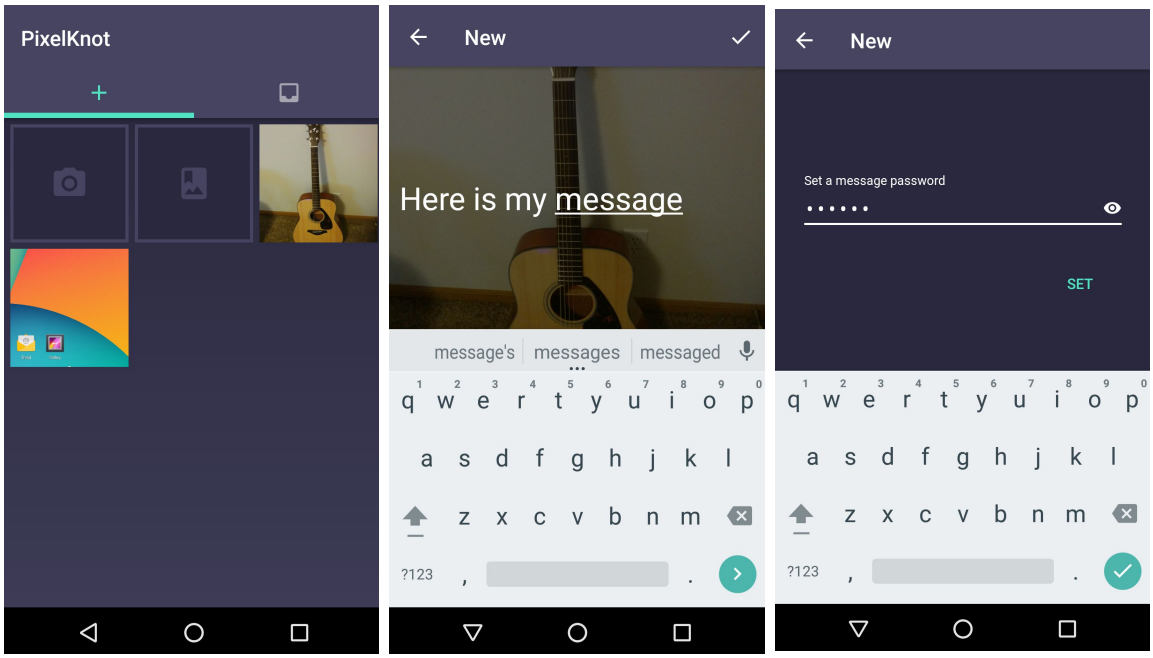
III. Android Steganogrpahy Applications and Reverse Engineering

Steganalysis applications in Android have certain common characteristics that can be exploited during the reverse engineering process. These characteristics can be used to reduce the scope of code analysis, and provide clues that lead to the code location of the core embedding algorithm. In this section, we first analyze the common characteristics among Android steganography applications, then describe the procedure and technical details of the reverse engineering process.

A. Common Characteristics of Steganography in Android

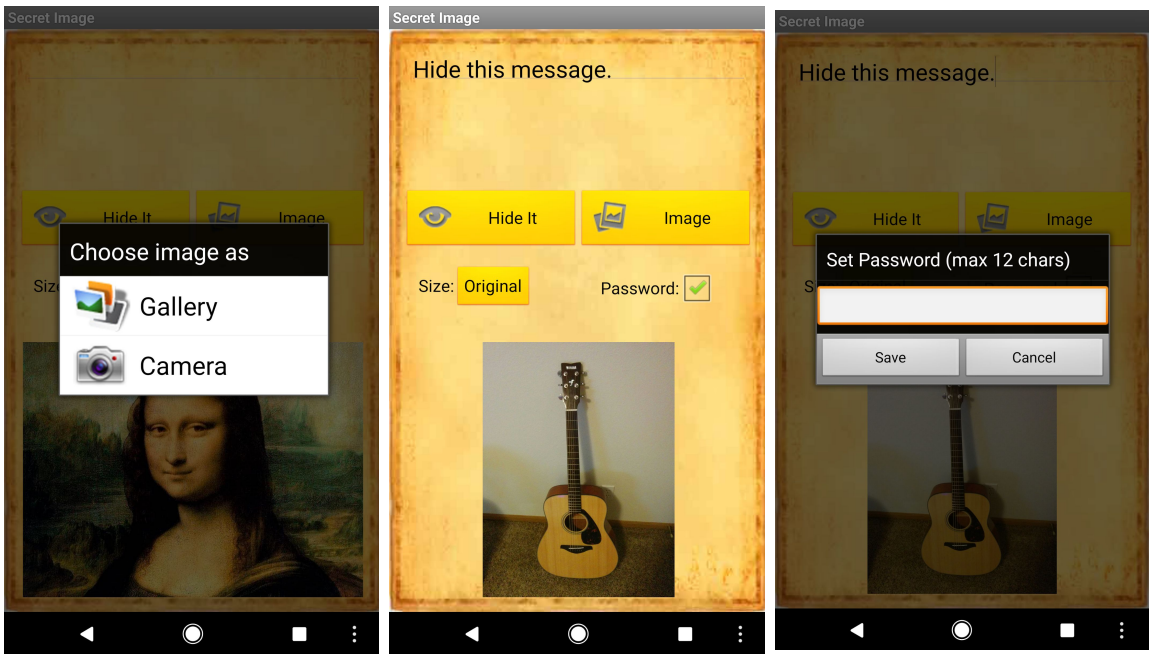
The first common characteristic of an Android steganography application is the user interface (UI) components. The minimum requirement for UI is to allow the user to: (1) select the cover image, (2) input the payload, and optionally (3) input a password. As such, the application must provide the corresponding UI components to enable these user interactions. As an example, Fig 1

and Fig 2 show the similarities of user input sequences between two apps: PixelKnot and Da Vinci Secret Image.



(a) select cover image (b) input the message to embed (c) input the password

Fig. 1: User input sequence for PixelKnot.



(a) select cover image (b) input the message to embed (c) input the password

Fig. 2: User input sequence for Da Vinci Secret Image.

Additionally, in order to allow the user to select an image or take a picture, the application is required to request the corresponding "permissions" in the program code and the manifest file, as shown in Fig 3 below.

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.CAMERA"/>
```

Fig. 3: Permission request code from the Android application manifest

Another common characteristic is the usage of image processing libraries. Although there are several available image processing libraries in Android, the image pixels will always be loaded into a Bitmap object that stores all the pixel values. Therefore, the embedding algorithm code will inevitably use an instruction that instantiates a Bitmap object, and method calls that access the Bitmap object, such as *Bitmap.getPixel(x,y)*.

B. The Reverse Engineering Process

We describe a general method to perform reverse engineering of steganography apps. Overall, the reverse engineering process involves three steps:

1) Extracting the application code
2) Locating the core embedding algorithm
3) Analyzing the embedding algorithm

We first use the reverse engineering tool *Apktool* to extract the application code along with the resource files, and then search for the code location of the core embedding algorithm. The reason for choosing *Apktool* over others is that *Apktool* provides the most accurate representation of the binary code. It does not attempt to transform or optimize the original binaries, and simply increases the code readability by a one-to-one mapping from DEX instructions to *Smali* instructions. Although more difficult to read than other instruction formats, *Smali* guarantees the integrity of the code from the extraction.

Locating the core embedding algorithm is a two-pronged approach. We first inspect the embedding workflow in the UI domain. To do this, we run the application on a test device and record the user input sequence during an embedding task. Using the Android UI debugging program UIAutomator [23], we look for the resource ID of each UI component in the input

sequence. With the resource IDs, we can then locate the *Smali* code of callback method for each UI component. These callback methods possibly contain the code for image processing, payload processing, and payload embedding.

However, due to the flexibility of Android UI programming, it is possible that the UI components have empty ID fields. Since Android allows authors to register callback methods for UI components created during runtime, resource ID is not needed. In this case, we search for the embedding algorithm using keywords. As previously mentioned, certain libraries and objects will most likely be used during embedding. Using keywords such as *BitmapFactory* and *Bitmap.getPixel(x,y)*, we can trace the execution flow and eventually locate the entry point of the embedding algorithm.

After the embedding algorithm code is located, we manually inspect the code to find the lines that perform the embedding. Generally, an embedding algorithm starts by defining the order in which the pixels are visited, called the *embedding path*. Next, the payload is divided into bits or bytes and then embedded in a certain way along the embedding path. Other embedding tasks such as payload encryption and random path generator also need to be analyzed. Since almost every steganography application has a unique way of embedding the bitstreams, the algorithm analysis is a process that varies case by case and relies on the experience of the analyst.

IV. Case Study on PixelKnot and Da Vinci Secret Image

In this section, we provide the detailed analysis of reverse engineering two Android apps, PixelKnot and Da Vinci Secret Image. These two Google Play Store apps have similar user interfaces and functionality. However, they have very different embedding processes underneath the user interface. We discuss their analysis results separately.

A. PixelKnot

PixelKnot [14] is an Android implementation of the academic steganography algorithm F5 [24], with some modifications. We examine PixelKnot's user interface by running the app on an Android test device (a Google Pixel), and we analyze the embedding algorithm code by reading its publicly available source code from Github [27]. To distinguish between PixelKnot's version of F5 and the academic version, we call the academic version of F5 as implemented in computer code *standard F5*.

Fig. 4 shows the work flow of PixelKnot's embedding process. PixelKnot takes three user inputs: the image selected for embedding; the payload (text message); and a password. It then produces a JPEG format output image, the stego image. F5 uses the *quantized Discrete Cosine Transform* space for embedding the bits.

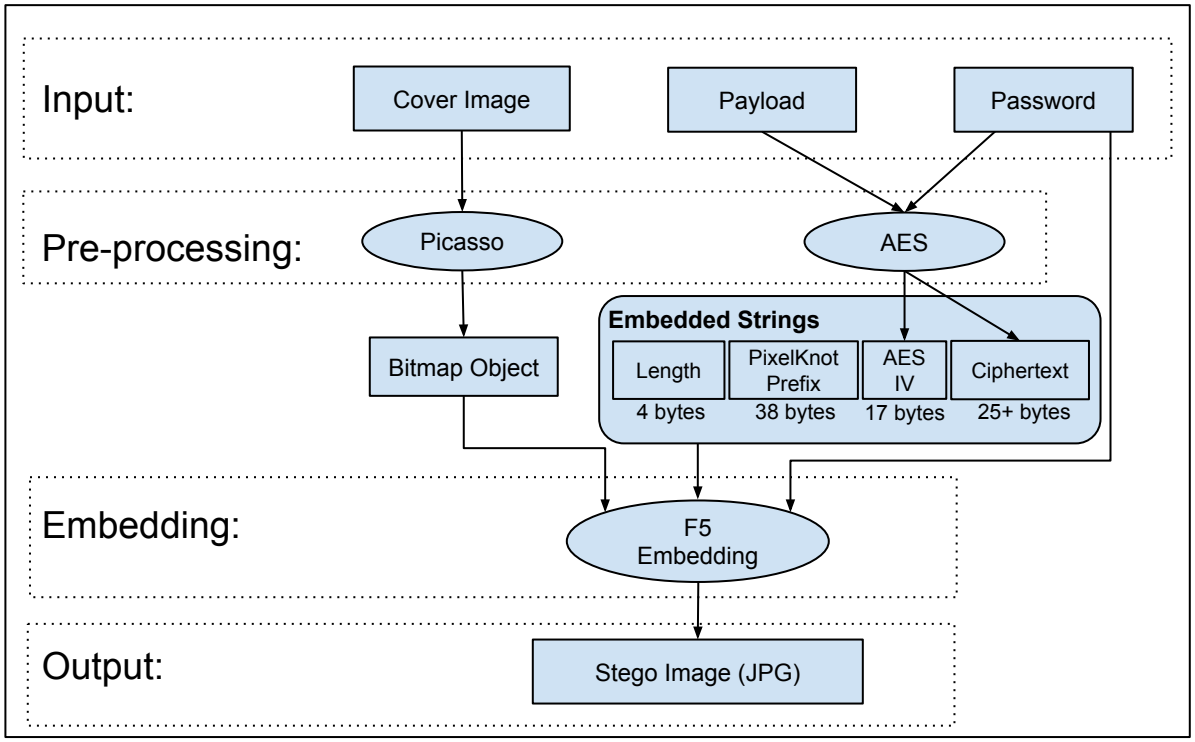


Fig. 4: Embedding work flow of PixelKnot.

The PixelKnot algorithm performs two preparatory steps before executing the bit embedding process. First, the input image is resized by downsampling if either its width or height exceeds 1280 pixels. In this case, the larger side is scaled to 1280 pixels, and the other side is scaled in proportion to the original dimensions. For example, a 1920*1280 input image will be downsampled to 1280*853 pixels in size. The resized image is then loaded into a Bitmap object, which is a matrix array of the pixel values. Second, the creation of the bit string ultimately embedded is a concatenation of four strings: a length string; a constant string; a string representing the initialization vector for the AES encryption; and ciphertext produced by encrypting the payload text using AES encryption. We describe each string next.

1) The length string indicates the number of bits of the ciphertext. The length string is 4 bytes long.

2) The constant string is 38 bytes long and consists of the following characters: *"—-* PK v 1.0 REQUIRES PASSWORD —-*"*.
3) The initialization vector (IV) is always 17 bytes long, and is a randomly-generated string used as part of the AES encryption that produces the ciphertext. It is stored in the image so that it can be used to extract the message later.
4) The last string is the AES-encrypted payload of the payload text input by the user.

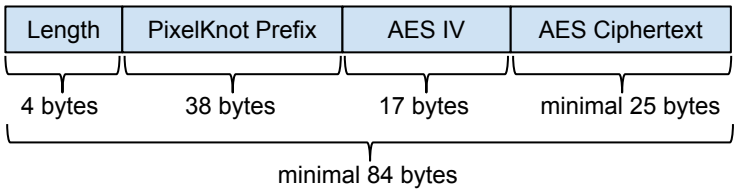


Fig. 5: Format of the embedded message payload in PixelKnot

As the ciphertext generated by PixelKnot's AES encryption has a minimum length of 25 bytes, the resulting bit string embedded into the input image has a minimum length of 84 bytes, as shown in Fig. 5.

The algorithm that PixelKnot uses to produce the ciphertext adds security over the standard F5 algorithm. First, an AES secret key is generated using the function *PBKDF2 with HMAC and SHA1*, and uses the first third of the password as the key and the second third of the password as salt. Second, using the *AES-GCM-NoPadding* cipher, the plaintext is encrypted with the AES secret key and a random initialization vector (IV). Therefore the IV needs to be stored in the image, as the ciphertext needs it for decryption. Finally, a pseudo-random pixel site visitation of the pixel sites in the image is generated using the last third of the password. This random path through the image visits a pixel value, and then embeds a bit there according to the F5 algorithm. The last part, spreading the bits that are embedded randomly around the image, ensures that even though a constant string is embedded, it can only be found if the password is known. Also, note that using the same password, the same input image, and the same payload text, that the ciphertext is different each time the app is run. This is because the IV, which is randomly generated, is different at each run of the app, thus producing a different ciphertext string each time the app is run. Thus, the security of PixelKnot's implementation of F5 depends largely upon the strength of the password.

In order to generate thousands of stego images to evaluate the stego detection programs'

effectiveness, we install PixelKnot on multiple Android emulators running on a computer to batch generate stego images. To verify that the emulator environment is identical to real Android devices when running PixelKnot, we devise a test to compare the stego images produced from an emulator with the stego images from a real device. Due to the randomness of the initialization vector (IV), even with the same plaintext message and password, PixelKnot produces different ciphertexts in different runs, which results in different stego images. Therefore, our verification test uses a slightly modified version of PixelKnot called PK.v1, which removes the AES encryption to remove the randomness. We install PK.v1 on two Android emulators and a Google Pixel phone. Given identical payload texts, identical cover images, and identical passwords to PK.v1 on the emulators and the Pixel, the stego images produced by each were identical. We perform this test 10 times, using 10 different combinations of images, payloads, and passwords. Once we verified that the emulator did indeed exactly mimic code running on the Pixel phone, we created a second version of the PixelKnot source code, called PK.v2, to efficiently generate large numbers of stego images. This version removes all the user interface portions from the original app, and adds functionalities such as saving an intermediate cover image and saving embedding stats including the embedding rate. PK.v2 reads input images from a folder, uses different passwords, different payloads, and different pre-determined embedding rates, to generate corresponding stego images. In this manner, we generated over 4000 stego images at the rate of about 100 images per minute.

B. Da Vinci Secret Image

Da Vinci Secret Image is a steganography application that uses a simpler embedding algorithm than PixelKnot. Due to the absence of available source code, we extracted the *Smali* code from its APK file using *Apktool*, and located its embedding algorithm code using the aforementioned two-pronged code locating approach. We then performed analysis on the target *Smali* code.

The Da Vinci Secret Image app provides similar functionalities to the user as PixelKnot. It allows the user to select a cover image, input the text to hide, and optionally enter a password. The user can also select one of a fixed number of image dimensions for the output image, including the option to maintain the original size. Depending on the selected size, the input image may be resized before the embedding process. While several different formats are supported for the input image, the stego output is always PNG (Portable Network Graphics) format.

A picture of the embedding process is shown in Fig. 6. The embedding is performed in the alpha channel of the PNG image, visiting pixel sites in a lexicographical manner from top left to bottom right. This is contrary to PixelKnot, where the pixel site visitation is random.

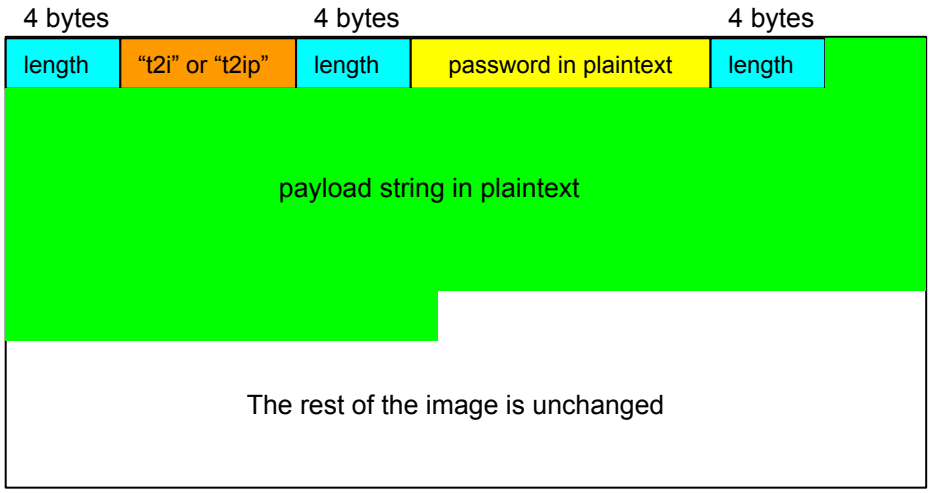


Fig. 6: Schematic of the embedding process in Da Vinci Secret Image.

The image is pre-processed to prepare for embedding. The input image file is decoded and loaded into a Bitmap object using the android API *BitmapFactory.decodeFile(path/to/image)*. If the user has chosen a size that is not the original, the Bitmap object is resized to match the target size.

Next, a series of strings are generated. Below is the format of each string:

1) The first “length string” indicates the number of bits in the string “t2i” or “t2ip,” depending on whether the user input a password. The length string is fixed at 4 bytes, or 32 bits. If a password is given by the user, then the length string consists of the bits 100000, preceded by 26 zeros (as there are 32 bits in the length string). If a password is not given by the user, then the length string consists of the bits 10100, preceded by 27 zeros.
2) The second string is the bit representation of “t2i” or “t2ip”. If a user does not input a password, then the string “t2i” is embedded; if a user does input a password, then the string “t2ip” is embedded. This string is always 4 bytes long.
3) The third string is also a 4 bytes long “length string,” indicating the length of the password.
4) The fourth string is the bit representation of the password, in plaintext.
5) The fifth string is another 4 bytes long “length string” indicating the length of the payload.
6) The final string is the payload in bit representation, in plaintext.

The remaining bits of the image are unchanged should the payload string be shorter than the remaining bits. The six strings are concatenated and then embedded into the alpha channel. The alpha channel can be viewed as a fourth 8-bit plane of the RBG color image in PNG format. The bit value "zero" of the string to embed is given the value 254 in the alpha channel. The bit value "one" is given the value 255 in the alpha channel. If the input image had information in the alpha channel and the original size is unchanged, information in the alpha would be overwritten. However, changing pixels in the alpha channel does not change at all the RGB values representing the image content, and thus the image scene is untouched.

Note that once the embedding process is known, it is straightforward to analyze a PNG image and determine if Da Vinci Secret Image produced it. First, inspection of the alpha channel for the characters "t2i" or "t2ip" in the fifth through eighth bytes location identifies this as a stego produced by Da Vinci, so the first 64 bits of the alpha channel serves as a signature. Second, the app uses the password only to verify that extraction of the payload can proceed, not to encrypt the payload. If an incorrect password is given, the app will not extract the payload. However, knowing information resides in the alpha channel, and once "t2i" or "t2ip" is observed, the length of the payload can be read and the payload extracted and reconstructed into plaintext.

Despite the similarities in user interface and functionality, the embedding process of Da Vinci is different from that of PixelKnot. Da Vinci uses a fixed embedding path as opposed to a random embedding path as in PixelKnot. Da Vinci embeds bits directly into the alpha channel, while PixelKnot embeds the message bits into the quantized DCT domain of JPEG. Most importantly, there is no randomness or encryption in Da Vinci. Our analysis of Da Vinci's algorithm reveals that its stego images have an easily detectable signature. Due to the embedding of a signature message and the absence of encryption and randomness, simply reading the first 64 pixels is enough to identify a Da Vinci stego image.

V. Performance Evaluation of Steg Detection Algorithms

In this section, we evaluate three existing steg detection programs that are publicly available: *StegDetect* by DC3, commercially available *Stego Hunt* [16], and freeware *StegDetect* by Provos [26], to answer the following research question:

- How effective are existing detection programs in detecting stego images generated from stego apps and other existing freeware code?

A. Experimental Setup

DC3 StegDetect is a software program developed to detect stego-related files on a computer. It has a GUI interface with several options to run it, including which programs to detect. It is a software program that can be applied to many different types of files, including executable files and stego images. We applied it to image files and executable files. It was last updated in the mid 2000s, and so does not contain signatures of programs created or updated past then. It uses signatures for detection, and attempts to extract a password, decrypt it, and extract the payload, if possible.

StegoHunt is commercial software by WetStone. It is purchased at an initial price, with yearly renewal of the license. It advertises StegoHunt as the "leading software tool for discovering the presence of data hiding activities," and it can "generate case specific reports for management or court presentation" and "identify suspect carrier files: program artifacts, program signatures, statistical anomalies." The web pages do not specify, but it is likely that the program uses hash tables for lookups, file signatures, and statistics to perform its analysis of files. There are 10 possible detection responses for a given scanned file, and it outputs results of the scanned files in a report.

StegDetect by Provos is a completely different software from DC3's StegDetect, developed by Dr. Neil Provos. It only accepts JPG images as input to scan, and is designed to detect stego images that are output by the three steganography programs: *jsteg* [27], *jphide* [28], and *outguess 0.13b* [29]. All three steg embedding programs output JPEG formatted images. If a file is detected as stego, the program then identifies the most likely embedding algorithm used.

We run the three detection programs on a set of images that contains both cover and stego images. A subset of the images are in PNG format while the rest are in JPG format. Due to the fact that *StegDetect* by Provos does not take PNG files as input, we excluded the PNG files for this particular program.

B. Detection Results

To produce stego images to run through each of the three detection programs, we first create a set of cover images. We use images that were acquired by a set of mobile phones as part of the authors' research lab [30], and create cover images in both PNG and JPG formats. Detailed information of the image data is shown in Table I. The test images are grouped into five sets, each indicating a different file type or embedding algorithm.

TABLE I: Overall Information of the Image Data

Image Data tested on Steg Detection				
Set Index	**File Type**	**Cover or Stego**	**Total #**	**Embedding Algorithm**
1	PNG	Cover	2090	(none)
2	JPG	Cover	1606	(none)
3	JPG	Stego	4818	PixelKnot
4	JPG	Stego	421	standard F5 Steganography
5	PNG	Stego	10	Camouflage

First, we run the detection programs on the cover images in Set 1 and Set 2. The detection results are shown in Table II. Note that StegoHunt identifies more than half of the cover PNG

TABLE II: Detection Results on the Cover Images

Image Data		**Detection Results of:**		
Set Index	**Total Number**	**Stego Hunt**	**DC3 StegDetect**	**Provos StegDetect**
1	2090	1304 Carrier Anomalies	0 suspicious	N/A
2	1606	0 anomalies	0 suspicious	380 stegos (24%)

images as having "anomalies." We suspect this may be due to the different type of file formatting to PNG these images received. Additionally, Provos StegDetect identified 24% of the cover JPGs as stego images.

Next, we test the detection programs on the stego images in Set 3 and Set 4. The stego images from Set 3 were generated from PixelKnot using scripts on Android emulators. (Note: This set of stego images is generated using different embedding rates; generally speaking, a longer payload means more bits changed, and that means detection can be easier. We do not discuss this further as it is beyond the scope of the paper.) The stego images in Set 4 were generated from the standard F5 steganography algorithm implemented on a desktop computer [25]. The detection result on

TABLE III: Detection Results of stego images from PixelKnot and standard F5

Image Data			**Detection Results of:**		
Set Index	**Total Number**	**Embedding Algorithm**	**Stego Hunt**	**DC3 StegDetect**	**Provos StegDetect**
3	4818	PixelKnot	0 anomalies	0 suspicious	1160 stegos (24%)
4	421	standard F5	399 Carrier Anomalies	421 marked as F5	223 stegos(53%)

these two sets is shown in Table III. Neither *StegoHunt* nor *DC3 StegDetect* properly detected a

single stego image from PixelKnot. Recall that PixelKnot was created around 2012, and thus is not in the DC3 StegDetect database. Here, *Provos StegDetect* correctly identified around 24% of the PixelKnot stego images. For the F5 stego images in Set 4, *StegoHunt* identified almost all as having "anomalies" but not "stego", while *DC3 StegDetect* properly identified all 421 stego images as being embedded with the standard F5 algorithm. However, *Provos StegDetect* identified only around 53% of the stego images properly, about the same as randomly guessing.

TABLE IV: Detection Results on the Camouflage Stego Images

Image Data			**Detection Results of:**		
Set Index	**Total Number**	**Embedding Algorithm**	**Stego Hunt**	**DC3 StegDetect**	**Provos StegDetect**
5	10	Camouflage	10/10: data appended past EOF	10/10: detected as Camouflage	N/A

Finally, using an older steganography software Camouflage [2], we created 10 stego images as the fifth image set. The detection result is shown in Table IV. Both *StegoHunt* and *DC3 StegDetect* correctly identified all 10 images as stego, with *StegoHunt* correctly responding that data was appended past the EOF, and *DC3 StegDetect* marking the images as from Camouflage. Additionally, *DC3 StegDetect* extracted the password and payload for the 10 stego images.

C. Discussions

In our experiments, the *DC3 StegDetect* outperforms the other two software packages on the data that we provided. *StegoHunt* identifies Camouflage stegos and anomalies in most standard F5 stegos, though not correctly as stego images. *DC3 StegDetect* identifies all F5 stegos, and identifies and extracts messages in all Camouflage stegos. Neither *StegoHunt* nor *DC3 StegDetect* identify PixelKnot stegos. *Provos StegDetect* has a high False Alarm Rate (24% as shown in Table II) and high Missed Detection Rate (75% as shown in Table III). Among the 223 stegos in image set 4 detected by *Provos StegDetect*, 219 were correctly identified as standard F5 stegos while the other 4 were incorrectly identified as Outguess and jphide. We observe that *Provos StegDetect* identifies all images tested here, cover or stego, with a rate between 25% and 50%, not a useful feature for steg detection.

Since both *StegoHunt* and *DC3 StegDetect* can identify steganography programs, we used those two programs to scan the two executables of standard F5 and Camouflage, and the source code of standard F5. Neither program was able to correctly identify any of those three files.

VI. Conclusion

Mobile phones apps for steganography are slowly becoming more available, and this research shows that some apps lend themselves to reverse engineering. We summarized the common characteristics of steganography programs on the Android platform and examined two Android steganography apps: PixelKnot and Da Vinci Secret Image.

Our analysis shows that, despite having similar user interfaces, the two apps have completely different embedding processes. PixelKnot is based on the academic algorithm, F5 steganography, hiding the payload in the quantized DCT domain and implementing anti-analysis measures such as encryption and randomness. Da Vinci Secret Image, on the other hand, was simple and straightforward to analyze. We revealed an easily detectable signature in this app. Without encryption or randomness, the app exhibited a signature that would properly identify its stego images. Other newer stego apps may also have their own signatures.

Finally, we showed that existing software was not adequate to identify stego images from the more recent steganography app PixelKnot. We believe that the field of steg detection for mobile steg apps has room to expand and improve.

VII. Acknowledgement

This work was partially funded by the Center for Statistics and Applications in Forensic Evidence (CSAFE) through Cooperative Agreement #70NANB15H176 between NIST and Iowa State University, which includes activities carried out at Carnegie Mellon University, University of California Irvine, and University of Virginia. We are grateful to the following undergraduate students for helping us acquire images for our database that are used in this experiment: Yiqiu Qian; Joseph Bingham; Chase Webb; and Mingming Yue.

References

[1] A. Burn, A. De Sélincourt *et al.*, "Herodotus. histories," 1955.

[2] Camouflage Software Inc., "Camouflage," http://camouflage.unfiction.com/, last accessed on 2017-12-04.

[3] "Jpegx," http://www.nerdlogic.org/jpegx/old/jpgx.html, 2001-2016, last accessed on 2017-12-04.

[4] Sky Juice Software, "Data stash," http://www.skyjuicesoftware.com/software/ds_info.html, last accessed on 2017-12-04.

[5] N. F. Johnson and S. Jajodia, "Exploring steganography: Seeing the unseen," *Computer*, vol. 31, no. 2, 1998.

[6] P. Alvarez, "Using extended file information (exif) file headers in digital evidence analysis," *International Journal of Digital Evidence*, vol. 2, no. 3, pp. 1–5, 2004.

[7] A. Cheddad, J. Condell, K. Curran, and P. Mc Kevitt, "Digital image steganography: Survey and analysis of current methods," *Signal processing*, vol. 90, no. 3, pp. 727–752, 2010.

[8] I.-S. Lee and W.-H. Tsai, "A new approach to covert communication via pdf files," *Signal processing*, vol. 90, no. 2, pp. 557–565, 2010.

[9] J. Fridrich and J. Kodovsky, "Rich models for steganalysis of digital images," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 3, pp. 868–882, 2012.

[10] F. Huang, J. Huang, and Y.-Q. Shi, "New channel selection rule for jpeg steganography," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 4, pp. 1181–1191, 2012.

[11] F. Djebbar, B. Ayad, K. A. Meraim, and H. Hamam, "Comparative study of digital audio steganography techniques," *EURASIP Journal on Audio, Speech, and Music Processing*, vol. 2012, no. 1, p. 25, 2012.

[12] M. M. Sadek, A. S. Khalifa, and M. G. Mostafa, "Video steganography: a comprehensive review," *Multimedia tools and applications*, vol. 74, no. 17, pp. 7063–7094, 2015.

[13] W. Mazurczyk, P. Szaga, and K. Szczypiorski, "Using transcoding for hidden communication in ip telephony," *Multimedia Tools and Applications*, vol. 70, no. 3, pp. 2139–2165, 2014.

[14] Guardian Project, "Pixelknot: Hidden messages," https://guardianproject.info/apps/pixelknot/, 2017.

[15] RADJAB, "Da vinci secret image," https://play.google.com/store/apps/details?id=jubatus.android.davinci, 2012.

[16] WetStone Technologies, "Stego hunt," https://www.wetstonetech.com/product/stegohunt/, 2017.

[17] S. Lyu and H. Farid, "Steganalysis using higher-order image statistics," *IEEE transactions on Information Forensics and Security*, vol. 1, no. 1, pp. 111–119, 2006.

[18] "Dalvik bytecode," https://source.android.com/devices/tech/dalvik/dalvik-bytecode, last accessed on 2017-09-17.

[19] R. Winsniewski, "Android–apktool: A tool for reverse engineering android apk files," 2012.

[20] "Smali," https://github.com/JesusFreke/smali/wiki, last accessed on 2017-09-17.

[21] B. Pan, "dex2jar," https://github.com/pxb1988/dex2jar, 2015.

[22] "Java decompiler," http://jd.benow.ca/, last accessed on 2017-09-17.

[23] Android Developers, "Ui automator," https://developer.android.com/training/testing/ui-automator.html, 2013.

[24] A. Westfeld, "F5 a steganographic algorithm," in *Information hiding*. Springer, 2001, pp. 289–302.

[25] "F5 steganography source code," https://code.google.com/archive/p/f5-steganography/, last accessed on 2017-12-04.

[26] N. Provos, "Stegdetect," http://www.outguess.org/detection.html, 1999-2014.

[27] "Jsteg," https://zooid.org/ paul/crypto/jsteg/, last accessed on 2017-09-17.

[28] "Jphide steganography," http://linux01.gwdg.de/ alatham/stego.html, 1999.

[29] N. Provos, "Outguess," http://www.outguess.org/, 1999-2014.

[30] CSAFE, "StegoDB: An image dataset for benchmarking steganalysis algorithms, Final Technical Report," Tech. Rep., June 2017.