Forensic Analysis of Android Steganography Apps

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

Chapter 16

FORENSIC ANALYSIS OF ANDROID STEGANOGRAPHY APPS

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

Abstract The processing power of smartphones supports steganographic algorithms that were considered to be too computationally intensive for handheld devices. Several steganography apps are now available on mobile phones to support covert communications using digital photographs.

This chapter focuses on two key questions: How effectively can a steganography app be reverse engineered? How can this knowledge help improve the detection of steganographic images and other related files? Two Android steganography apps, PixelKnot and Da Vinci Secret Image, are analyzed. Experiments demonstrate that they are constructed in very different ways and provide different levels of security for hiding messages. The results of detecting steganography files, including images generated by the apps, using three software packages are presented. The results point to an urgent need for further research on reverse engineering steganography apps and detecting images produced by these apps.

Keywords: Image forensics, steganography, steganalysis, Android apps

1. Introduction

The field of covert communications has a long history. Message encryption, called cryptography, is a well-known method for secret communications, but its limitation is that the transmission of encrypted messages is not kept secret. Steganography, on the other hand, attempts to send a message while hiding the fact that it is being transmitted, essentially evading the detection of the secret communication itself. The word steganography originates from Greek, meaning "covered writing." The first evidence of steganography dates back to late sixth century BC Greece: Herodotus [11] describes how Histiaeus sent his slave to the Io-

nian city of Miletus with a hidden message to incite a revolt against the Persian king. The slave's head was shaved, a message was tattooed on his scalp and his hair was allowed to grow back. After his hair had concealed the message, the slave was sent to Aristagoras, the city's regent. Aristagoras had the slave's head shaved and read the secret message. Modern versions of steganographic communications include invisible ink and microdots. Digital versions of steganographic messaging are now provided by software on computers as well as by smartphone apps.

Digital steganography hides a message or payload in the form of bits in a cover medium, also represented by bits, so as to not arouse suspicion of the hidden content. The cover file is combined with the payload to produce a "stego" file. The stego file is transmitted to the intended recipient, who extracts the hidden payload. A key is sometimes involved in steganographic communications.

The fundamental question is: How can one change the bits of a cover medium, such as a digital photograph, PDF document, digital audio file or video file, to represent the payload bits and then have the recipient successfully extract the payload after the stego version is received?

Hiding a payload in a digital photograph may be accomplished by appending it after the end-of-file (EOF) marker in a JPG image [23, 25, 29], in the color palette of a GIF image [14], in the EXIF header of an image [1, 5], in a PDF file [15], in the lower bits of a non-compressed RGB or grayscale image [8] or in the quantized discrete cosine transform coefficients of a compressed JPG image [12]. Digital audio [6] and video [22] files can also be used to hide payloads, as well as TCP/IP packets [17].

The discovery of stego-related files is called "steg detection." The existence of stego executables and related files on a system may indicate that steganography was used, so steg detection includes the detection of ancillary files that do not contain payloads. Patterns in stego files such as an embedded signature or statistical properties can also be exploited in steganalysis. Signature-based detection of a stego or ancillary file, based on specific characters or, perhaps, locations of added information, requires signatures and computer code that opens the file and searches for signatures. Statistics-based detection employs statistical measures of a suspected stego file and searches for abnormalities that indicate steganography. Another type of steg detection engages machine learning algorithms that do not depend on signatures written explicitly into stego files.

Although a mobile phone app makes steganography very easy to use, the detection of stego images produced by mobile phone apps has not as yet been discussed in the literature. Mobile apps for iOS and Android

phones can conceal a text payload in a photograph stored on the phone or acquired using the camera. Certain apps enable another file (e.g., image file) to be hidden in the cover image. However, some of the roughly 30 available steganography apps are unstable and may crash when using certain cover photographs, large payloads or specific mobile phone models.

The code used in the bit embedding process, where the bits of a cover image are changed to capture payload bits, varies according to the apps. In addition, not all apps have open-source code, which makes it difficult to reverse engineer their code. Thus, signature-based investigations of these apps are difficult, if not impossible, when signatures do not exist. Machine-learning-based detection may be more reliable than signature-based detection, but it requires substantial image data for training and classification.

In principle, applying machine learning to detect stego images from a mobile phone camera is no different from the "classical academic setting." In this classical setting, steganography or steganalysis is performed on a known set of image data, where the embedding algorithm is completely known and machine learning detection algorithms can be applied. Many academic steganography algorithms demonstrate the difficulty in detecting new embedding algorithms using known "best detection" algorithms (typically machine learning algorithms). In the case of steganalysis, the goal is to show that a new algorithm has clear performance advantages over existing algorithms. This chapter does not present any steganalysis results involving machine learning applications, although it includes a short description for completeness. The focus is on software that forensic practitioners can use to detect the existence of steganography.

This chapter discusses reverse engineering efforts on two Android apps, PixelKnot [10] and Da Vinci Secret Image [21]. The results underpin a procedure for generating a large quantity of stego images on a computer using PixelKnot code without having a human enter information manually using the mobile phone app. This chapter also presents the results of the first publicly-available evaluation of steg detection tools.

2. Related Work

This section discusses methods for detecting steganography payloads hidden in image files. Also, it discusses tools used for reverse engineering Android apps.

2.1 Steg Detection Approaches

Steg detection has two principal use cases: (i) discrimination of a normal image from a stego image; and (ii) identification of a file that can be associated with a steganographic process. While steganalysis commonly refers to the first use case, the second use case covers the identification of executable files and other non-image files involved in producing stego images. Digital forensic practitioners are interested in both use cases.

There are three basic types of steg detection approaches: (i) signature-based detection; (ii) hash-based detection: and (iii) machine-learning-based detection:

- **Signature-Based Detection:** Signature-based detection identifies possible signatures that a steganography program writes to an output stego image. Following this, stego files are detected using the signatures.

  Stego signatures exist in various forms. A program could embed the same fixed bit string along with a payload; or, the embedding path could visit the same pixel locations in the same order regardless of payload content, leaving a repeated pattern in the stego file.

  The commercial tool, StegoHunt [28], and the academic tool, StegDetect, perform signature-based steg detection. If the signature of the steganography program is known, then a signature-based approach can accurately identify stego images and, possibly, extract the hidden payloads. However, this approach requires continual updating of the detection code. This is because a change to the steganography program produces a different signature from the previous one, resulting in the failure to detect stego files created with the new program.

- **Hash-Based Detection:** Hash-based steg detection involves the identification of a previous, identical stego file, such as an image identified as child pornography. In this scenario, copies of the stego image that hide the payload are available; thus, all the stego images are identical in a bit-by-bit comparison and have identical hashes. This enables comparisons of the hash values of unknown images against a database of hash values of known stego images. New images that need to be analyzed are only required to have their hash values compared against the hash values of known stego images in the database.

- **Machine-Learning-Based Detection:** Machine-learning-based steg detection is a more complex task. In theory, a machine learning classifier can be constructed to identify an unknown stego image, if training data is available along with other caveats. The details of this type of approach are omitted because they are beyond the scope of this research. However, two machine-learning-based classifiers are discussed in [8, 16]. They require large amounts of training data – 700 to 6,000 cover images, the same number of corresponding stego images and a representative feature set and classifier. With these items and enough computing power, it is possible to create a machine classifier that detects stego images.

2.2 Reverse Engineering Android Applications

The approach for detecting stego images produced by an Android app involves the inspection of the app code. This so-called reverse engineering task attempts to understand the logic and other code details with the ultimate goal of exploiting certain characteristics of how the code processes images.

Reverse engineering is commonly used for program analysis. Analyzing an Android app often requires a reverse engineering tool that converts application binaries (APK) to a human-readable format. Android apps are developed in the Java programming language and compiled to Dalvik bytecode [3], which is similar to Java bytecode. The Dalvik bytecode is then encoded and written to a DEX file in the APK.

Several tools are available for extracting and decoding DEX files from an Android APK, and recreating the app code in the source or intermediate code format. Apktool [24] is designed for reverse engineering Android APK files; it decodes a DEX file to an intermediate code format called Smali [9]. The tool can also decode resource XML files, including the graphical interface definition and manifest file. Although Apktool does not translate DEX files into Java code, it provides accurate representations of binaries by avoiding translation loss.

The `dex2jar` tool [18] converts DEX to Java bytecode. Java Decompiler [13] can then be used to decompile the Java bytecode into Java source code, with possible loss of metadata and certain irreversible DEX code blocks. Using `dex2jar` and Java Decompiler in combination makes it possible to recreate the Java source code from an APK file. However, due to inconsistencies in Java Decompiler, the output source code can only be used as a reference for application analysis.

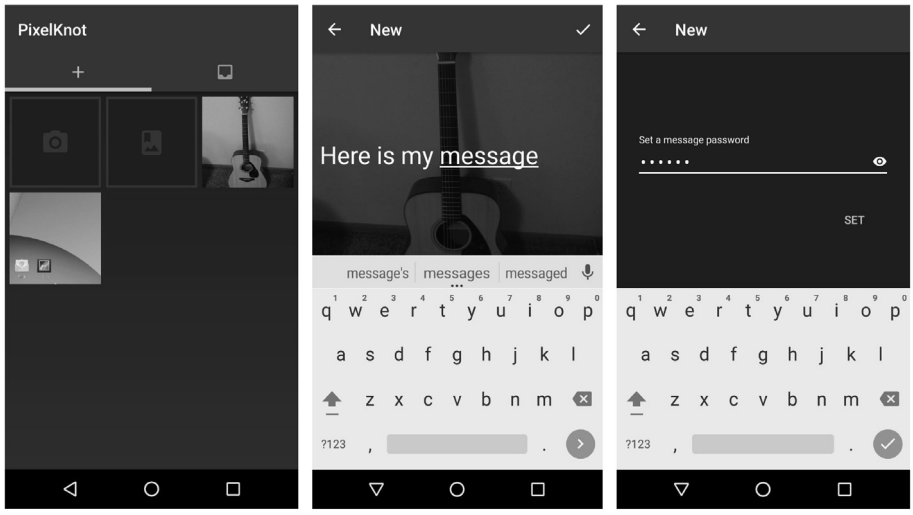

Figure 1. User input sequence for the PixelKnot app.

3. Android App Reverse Engineering

Steganography apps developed for Android devices have certain common characteristics that can be exploited during the reverse engineering process. These characteristics can be leveraged to reduce the scope of code analysis and provide clues that help locate the core embedding algorithm. This section analyzes the common characteristics of Android steganography apps and describes the technical details involved in reverse engineering the apps.

3.1 Common Characteristics

The first common characteristic of Android steganography apps is the user interface components. At a minimum, a user interface should enable a user to: (i) select a cover image; (ii) input a payload; and (optionally) (iii) input a password. As such, an app must provide the user interface components to enable these interactions. Figures 1 and 2 show the similarities of the user input sequences (select a cover image, input a message to embed and input a password) for the PixelKnot and Da Vinci Secret Image apps, respectively.

Additionally, as shown in Figure 3, in order to enable a user to select an image or take a picture, the app must request the corresponding "permissions" in the program code and manifest file.

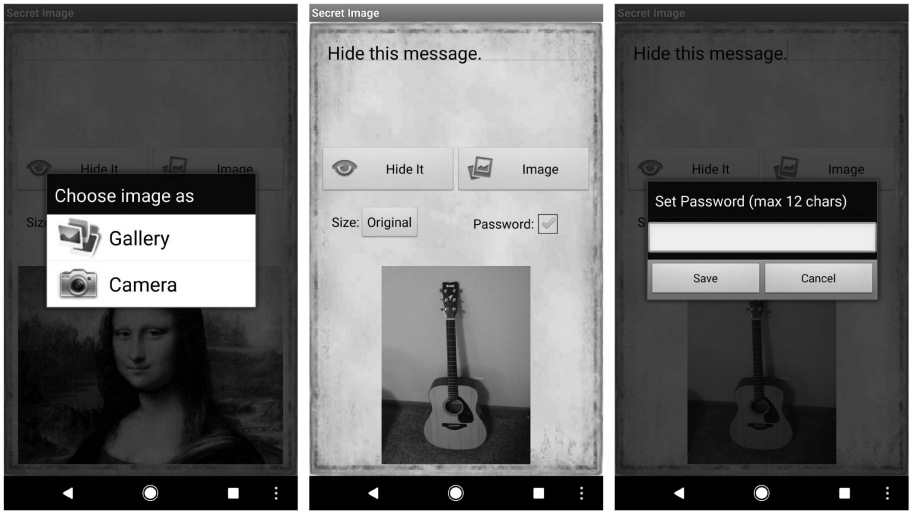

Figure 2. User input sequence for the Da Vinci app.

```
<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"/>
```

Figure 3. Permission request code from the Android app manifest.

Another common characteristic is the use of image processing libraries. Although there are several Android image processing libraries, the image pixels are always loaded into a bitmap object that stores all the pixel values. Therefore, the embedding algorithm always has an instruction that instantiates a bitmap object and a method call that accesses the bitmap object, e.g., `Bitmap.getPixel(x,y)`.

3.2 Reverse Engineering Process

Reverse engineering the Android steganography apps involved three steps:

- **Step 1. Extracting the App Code:** In this step, Apktool was used to extract the app code along with the resource files in order to search for the location of the core embedding algorithm. Apktool was selected over other tools because it provides the most accurate representation of binary code. It does not attempt to transform or optimize the original binaries, and increases code readability using a one-to-one mapping from DEX instructions to Smali instructions.

Although Smali is more difficult to read than other instruction formats, it guarantees the integrity of the code.

- **Step 2. Locating the Core Embedding Algorithm:** Locating the core embedding algorithm involved two parts. The first part involved an inspection of the embedding workflow in the user interface domain. This was accomplished by executing the app on a test device and recording the user input sequence during the embedding task. Next, the Android debugger UIAutomator [2] was used to search for the resource ID of each user interface component in the input sequence. The resource IDs were then used to locate the Smali code of the callback method for each user interface component. The callback methods may contain the code for image processing, payload processing and payload embedding.

  However, due to the flexibility of Android user interface programming, the user interface components could have empty ID fields. Since Android enables authors to register the callback methods for user interface components created at runtime, the resource IDs were not required. Therefore, the search for the embedding algorithm employed keywords. As mentioned above, certain libraries and objects would most likely be used during embedding. Keywords such as `BitmapFactory` and `Bitmap.getPixel(x,y)` could be employed to trace the execution flow and eventually locate the entry point of the embedding algorithm.

- **Step 3. Analyzing the Embedding Algorithm:** After the embedding algorithm code was located, it was inspected manually to find the lines of code that perform the embedding. Generally, an embedding algorithm starts by defining the order in which pixels are visited; this is called the embedding path. Next, the payload is divided into bits or bytes that are embedded in a certain manner along the embedding path. Other embedding tasks such as payload encryption and random path generation also must be analyzed. Since almost every steganography app has a unique way of embedding a bit stream, embedded algorithm analysis varies based on the app and relies greatly on the experience of the analyst.

4. Case Study

This section presents detailed results of the reverse engineering efforts on the PixelKnot and Da Vinci Secret Image Android apps. These two Google Play Store apps have similar user interfaces and functionality. However, they employ very different embedding processes.

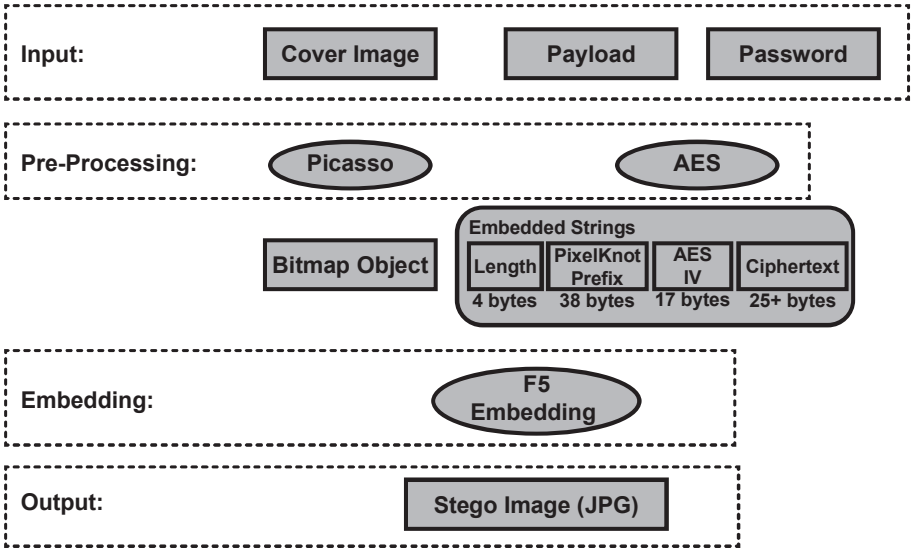

Figure 4. PixelKnot embedding process workflow.

4.1 PixelKnot

PixelKnot [10] is an Android implementation of the F5 steganography algorithm [27] with some modifications. The PixelKnot user interface was examined by running the app on an Android test device (Google Pixel phone). The embedding algorithm code was downloaded from Github [9]. The academic version of F5 is distinguished from the PixelKnot version of F5 by calling it standard F5.

Figure 4 shows the workflow of the PixelKnot embedding process. PixelKnot has three user inputs: (i) cover image; (ii) payload (text message); and (iii) password. It produces the stego image in the JPG format. The F5 algorithm uses the quantized discrete cosine transform technique to embed bits.

The PixelKnot algorithm performs two preparatory steps before executing the bit embedding process. First, the input image must be resized by downsampling if its width or height exceed 1,280 pixels. In this case, the larger side is scaled to 1,280 pixels and the other side is scaled in proportion to the original dimensions. For example, a $1,920 \times 1,280$ input image is downsampled to $1,280 \times 853$ pixels. The resized image is then loaded into a bitmap object, which is a matrix array of the pixel values. In the second step, the bit string to be embedded is created by concatenating four strings: (i) length string; (ii) constant string; (iii)

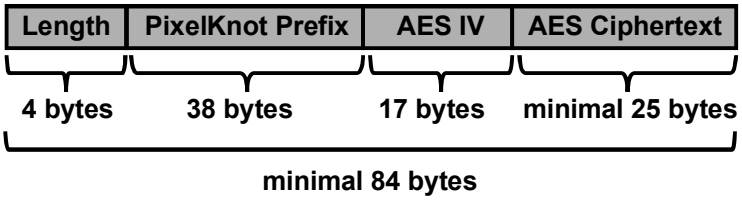

Figure 5. Format of the PixelKnot embedded message payload.

initialization vector for AES encryption; and (iv) ciphertext produced by encrypting the payload text using AES encryption:

- **Length String:** The length string indicates the number of bits in the ciphertext. It is 4 bytes long.
- **Constant String:** The constant string is 38 bytes long and has the characters: ‘‘`----* PK v 1.0 REQUIRES PASSWORD ----*`’’.
- **Initialization Vector:** The initialization vector is always 17 bytes long. It is a randomly-generated string used by AES to produce the ciphertext. The initialization vector is stored in the image so that it can be used to extract the message later.
- **AES-Encrypted Payload:** The payload is the AES ciphertext of the payload text input by the user.

The ciphertext generated by PixelKnot's AES encryption has a minimum length of 25 bytes. Therefore, the bit string embedded in the input image has a minimum length of 84 bytes (Figure 5).

The algorithm used by PixelKnot to produce ciphertext adds security over and above the standard F5 algorithm. First, an AES key is generated using PBKDF2 with HMAC and SHA1, where the first third of the password is the key and the second third of the password is the salt. Next, the AES-GCM-NoPadding cipher is used to encrypt the plaintext with the AES key and a random initialization vector; the initialization vector is stored in the image because it is needed to decrypt the ciphertext. Finally, a pseudorandom visitation of the pixel sites in the image is generated using the last third of the password. The random path through the image visits a pixel and embeds a bit at the site using the F5 algorithm. The random spreading of the embedded bits around the image ensures that even if a constant string is embedded, it can only be found if the password is known. Note that even if the same password, input image and payload text are used, different ciphertext is generated each time the app is run. This is because the initialization vector, which

is randomly generated, is different for each execution of the app. Thus, the security of the PixelKnot implementation of F5 depends largely on the strength of the password.

Thousands of stego images had to be generated in order to evaluate the effectiveness of the evaluated stego detection programs. This was accomplished by installing PixelKnot on multiple Android emulators running on a computer to batch-generate stego images. To verify that the emulator environment was identical to a real Android device when running PixelKnot, a test was devised that compared the stego images produced by an emulator with the stego images from a real device. Due to the randomness of the initialization vector, even with the same plaintext message and password, PixelKnot produces different ciphertext in different runs, resulting in different stego images. Therefore, the verification test used a slightly modified version of PixelKnot called PK.v1, which disabled AES encryption to eliminate randomness.

PK.v1 was installed on two Android emulators and a Google Pixel phone. Identical stego images were obtained upon providing identical cover images, payload text and passwords to PK.v1 on the emulators and on the Pixel phone. This test was performed ten times using ten different combinations of images, payloads and passwords.

After it was verified that the emulator exactly mimicked code running on the Pixel phone, a second version of the PixelKnot source code, called PK.v2, was created to efficiently generate large numbers of stego images. PK.v2 removed all the user interface portions from the original app while adding functionality such as saving an intermediate cover image and saving embedding stats, including the embedding rate. PK.v2 read input images from a folder and used different passwords, payloads and predetermined embedding rates to generate the corresponding stego images. This procedure generated more than 4,000 stego images at the rate of about 100 images per minute.

4.2 Da Vinci Secret Image

Da Vinci Secret Image is a steganography application that uses a simpler embedding algorithm than PixelKnot. Because its source code is not available to the public, Smali code was extracted from its APK file using Apktool. The embedding algorithm code was then located using the two-step approach described above. Following this, analysis was performed on the target Smali code.

The Da Vinci Secret Image app provides similar functionality as PixelKnot. It enables a user to select a cover image, input the payload text and, optionally, enter a password. The user can also select one of a fixed

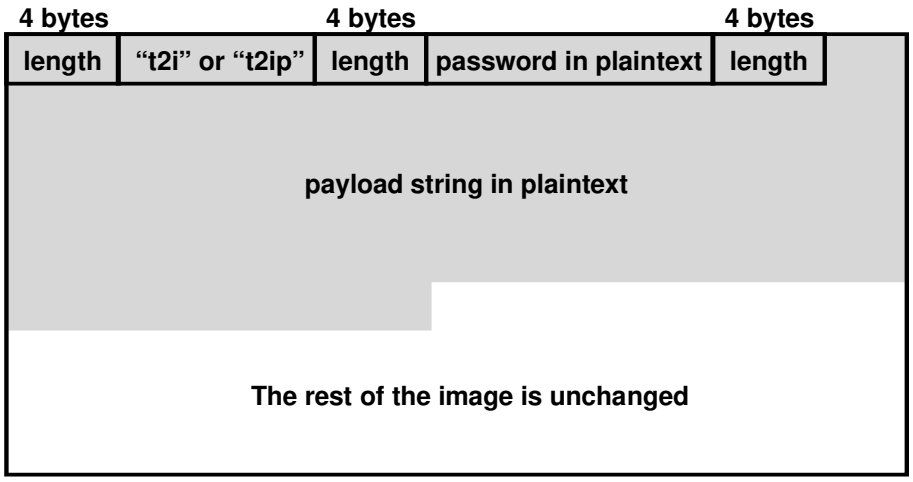

Figure 6. Embedding process of the Da Vinci Secret Image app.

number of image dimensions for the output image, including maintaining the original image 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 in the PNG format.

Figure 6 shows the embedding process of the Da Vinci Secret Image app. The embedding is performed in the alpha channel of the PNG image. During the embedding process, pixel sites are visited in a lexicographical manner from top left to bottom right. This is very different from PixelKnot, where the pixel site visitation is random.

The image is first pre-processed to prepare for the 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 selects a size that is different from the original size, the bitmap object is adjusted to match the target size.

Next, a series of six strings are generated:

- **String 1:** This string indicates the number of bits in the string `t2i` or `t2ip`, depending on whether or not the user entered a password. The string is always 4 bytes long. If a password was supplied by the user, then the string comprises the bits `100000` preceded by 26 zeros (because there are 32 bits in the length string); otherwise, the string comprises the bits `10100` preceded by 27 zeros.
- **String 2:** This string is the bit representation of `t2i` or `t2ip`. If a password was supplied, then the string `t2i` is embedded; otherwise, `t2ip` is embedded. This string is always 4 bytes long.

- **String 3:** This string, which is always 4 bytes long, indicates the length of the password.
- **String 4:** This string contains the bit representation of the plaintext password.
- **String 5:** This string, which is always 4 bytes long, indicates the length of the payload.
- **String 6:** This string contains the bit representation of the plaintext payload.

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 the fourth 8-bit plane of the RBG color image in the PNG format. The bit value "zero" of the string to be embedded is given the value 254 in the alpha channel whereas the bit value "one" is given the value 255. If the input image contains information in the alpha channel and the original size is unchanged, then information in the alpha is overwritten. However, changing pixels in the alpha channel does not change the RGB values representing the image content and, thus, the image scene is untouched.

After the embedding process is known, it is relatively straightforward to analyze a PNG image and determine if it was produced by the Da Vinci Secret Image app. First, the alpha channel is inspected for the characters `t2i` or `t2ip` in bytes 5–8. This identifies the stego file as being produced by Da Vinci, so the first 64 bits of the alpha channel serves as a signature.

The app uses the password only to verify that the extraction of the payload can proceed, not to encrypt the payload. If an incorrect password is provided, the app does not extract the payload. However, since it is known that information resides in the alpha channel, upon observing the characters `t2i` or `t2ip`, the length of the payload can be read and the payload extracted and reconstructed into plaintext.

Despite the similarities in their user interfaces and functionality, Da Vinci and PixelKnot employ very different embedding processes. Da Vinci uses a fixed embedding path whereas PixelKnot uses a random embedding path. Da Vinci embeds bits directly into the alpha channel whereas PixelKnot embeds bits into the quantized discrete cosine transform domain of JPG. Most importantly, Da Vinci neither uses encryption nor randomness. The analysis of the Da Vinci algorithm reveals that its stego images have easily detectable signatures. Due to the embedding of

a signature message and absence of encryption and randomness, merely reading the first 64 pixels is enough to identify a Da Vinci stego image.

5. Performance Evaluation

This section presents the results of evaluating three steganography detection programs: (i) StegDetect (from DC3); (ii) StegoHunt [28] (commercially-available); and (iii) StegDetect (by Provos) [20] (freeware). The evaluation sought to assess the effectiveness of the programs at detecting stego images generated by various Android apps.

5.1 Experimental Setup

StegDetect from DC3 is a software program that detects stego-related files on a computer. It has a graphical user interface that provides several options, including identifying the programs to be detected. The program can be applied to several types of files, including executable files and stego images. In the experiments, StegDetect was applied to image files and executable files. It uses signatures for detection, and attempts to extract a password, decrypt it and extract the payload. StegDetect was last updated in the mid 2000s, so it does not contain the signatures of new and updated programs.

StegoHunt, commercial software from WetStone [28], is advertised as the "leading software tool for discovering the presence of data hiding activities." It can "generate case specific reports for management or court presentation" and "identify suspect carrier files: program artifacts, program signatures, statistical anomalies." No details are provided, but it appears that the program uses hash tables for lookups and file signatures and statistics to perform its analysis. The software has ten possible detection responses for a given file and the results are provided in a report.

StegDetect by Provos [20] only accepts JPG images as input. It is designed to detect stego images produced by three steganography programs: (i) `jsteg` [26]; (ii) `jphide` [26]; and (iii) `outguess 0.13b` [19]. All three steg embedding programs output JPG images. If a file is detected as containing steganography, then StegDetect proceeds to identify the most likely embedding algorithm used.

In the experiments, the three detection programs were executed on cover and stego images. The images were in the JPG and PNG formats. Since StegDetect cannot handle PNG files, these files were not used when evaluating the program.

Table 1. Image sets used in the evaluation.

	Image Format	Image Type	Image Quantity	Embedding Algorithm
Set 1	PNG	Cover	2,090	None
Set 2	JPG	Cover	1,606	None
Set 3	JPG	Stego	4,818	PixelKnot
Set 4	JPG	Stego	421	Standard FS
Set 5	PNG	Stego	10	Camouflage

5.2 Detection Results

A set of cover images was first created. The images were acquired using a set of mobile phones [4]. The cover images were created in the JPG and PNG formats.

Table 1 describes the image sets used in the evaluation. The test images were grouped into five sets, each indicating a different file type or embedding algorithm.

Table 2. Detection results for cover images.

	Image Quantity	StegoHunt	StegDetect DC3	StegDetect Provos
Set 1	2,090	1,304 Carrier Anomalies	0 Suspicious	N/A
Set 2	1,606	0 Anomalies	0 Suspicious	380 Stegos (24%)

First, the three detection programs were applied to the cover images in Sets 1 and 2. Table 2 presents the detection results. Note that StegoHunt identified more than half of the cover PNG images as having "anomalies," which may be due to the different type of file formatting applied to produce the PNG images. StegDetect by Provos identified 24% of the cover JPGs images as stego images.

Next, the detection programs were tested on the stego images in Sets 3 and 4. The stego images from Set 3 were generated by PixelKnot using scripts executing on Android emulators. Note that this set of stego images was generated using different embedding rates. Generally, a longer payload means that more bits were changed, which makes detection easier; this topic is not discussed further because it outside the scope of this research. The stego images in Set 4 were generated by the standard F5 steganography algorithm executing on a desktop computer [7].

Table 3. Detection results for PixelKnot and Standard F5 stego images.

	Image Quantity	Algorithm	StegoHunt	StegDetect DC3	StegDetect Provos
Set 3	4,818	PixelKnot	0 Anomalies	0 Suspicious	1,160 Stegos (24%)
Set 4	421	Standard F5	399 Carrier Anomalies	421 Marked as F5	223 Stegos (53%)

Table 3 shows the detection results for Sets 3 and 4. Neither StegoHunt nor the DC3 StegDetect properly detected a single stego image created using PixelKnot. Note that PixelKnot was created around 2012 and, thus, is not in the DC3 StegDetect database. However, Provos StegDetect correctly identified around 24% of the PixelKnot stego images without, of course, the correct identifying algorithm (because the F5 algorithm was not one of the three labeled algorithms). In the case of the standard F5 stego images in Set 4, StegoHunt identified almost the images as having anomalies, but not as stego. In contrast, DC3 StegDetect properly identified all 421 stego images as being embedded by the standard F5 algorithm. However, Provos StegDetect identified only 53% of the stego images correctly, about the same percentage as random guessing.

Table 4. Detection results for Camouflage stego images.

	Image Quantity	Algorithm	StegoHunt	StegDetect DC3	StegDetect Provos
Set 5	10	Camouflage	0 out of 10 Data added after EOF	10 out of 10 Detected as Camouflage	N/A

For the final set of experiments, the older Camouflage steganography software [25] was used to create ten stego images (Set 5). Table 4 shows the detection results. Both StegoHunt and DC3 StegDetect correctly identified all ten images as stego, with StegoHunt correctly warning that data was appended past the end-of-file marker and DC3 StegDetect recognizing the images as having been created by Camouflage. Additionally, DC3 StegDetect extracted the passwords and payloads for all ten stego images.

5.3 Discussion

In the evaluation, DC3 StegDetect performed better than StegoHunt and Provos StegDetect on the image sets. StegoHunt identified Camouflage stegos and detected anomalies in most standard F5 stegos, but they were not detected correctly as stego images. DC3 StegDetect identified all the F5 stegos, and also identified and extracted the messages in all the Camouflage stegos. Neither StegoHunt nor DC3 StegDetect identified PixelKnot stegos. As shown in Table 2, Provos StegDetect had a high false alarm rate of 24% and a high missed detection rate of 75% (Table 3). Of the 223 stegos in Set 4 that were detected by Provos StegDetect, 219 were correctly identified as standard F5 stegos while the other four were incorrectly identified as Outguess and `jphide`. Finally, Provos StegDetect identified the tested (cover and stego) images with a rate between 25% and 50%, which is rather poor.

Since StegoHunt and DC3 StegDetect can be employed to identify steganography programs, they were also used to scan the standard F5 and Camouflage executables and the source code of standard F5. However, neither program was able to correctly identify any of the files.

6. Conclusions

This research has analyzed the common characteristics and key features of the well-known PixelKnot and Da Vinci Secret Image apps for Android devices. The analysis has revealed that, despite having similar user interfaces, the two apps have completely different embedding processes. PixelKnot is based on the F5 steganography algorithm that hides payloads in the quantized discrete cosine transform domain and implements anti-analysis measures such as encryption and randomness. Da Vinci Secret Image, on the other hand, is simple and straightforward to analyze. Since it does not employ encryption or randomness, the Da Vinci Secret Image app exhibited a signature that readily identifies its stego images. Other newer stego apps may also have their own signatures.

The evaluation has demonstrated that current steganography detection software is inadequate at identifying stego images created by PixelKnot, which is a relatively recent steganography app. Clearly, detecting steg files created by mobile steg apps is in its infancy, but it is an interesting and most important research area.

Acknowledgements

This research was partially supported by the Center for Statistics and Applications in Forensic Evidence (CSAFE) through Cooperative Agreement No. 70NANB15H176 between NIST and Iowa State University, which includes research conducted at Carnegie Mellon University, University of California Irvine and University of Virginia. The authors are grateful to Yiqiu Qian, Joseph Bingham, Chase Webb and Mingming Yue for their assistance in generating the images used in this research.

References

[1] P. Alvarez, Using extended file information (EXIF) file headers in digital evidence analysis, *International Journal of Digital Evidence*, vol. 2(3), 2004.

[2] Android Developers, UI Automator (`developer.android.com/training/testing/ui-automator.html`), 2018.

[3] Android Open Source Project, Dalvik Bytecode (`source.android.com/devices/tech/dalvik/dalvik-bytecode`), 2018.

[4] Center for Statistics and Applications in Forensic Evidence, StegoDB: An Image Dataset for Benchmarking Steganalysis Algorithms, Final Technical Report, Iowa State University, Ames, Iowa, 2017.

[5] A. Cheddad, J. Condell, K. Curran and P. McKevitt, Digital image steganography: Survey and analysis of current methods, *Signal Processing*, vol. 90(3), pp. 727–752, 2010.

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

[7] F5-Steganography Project, F5 Steganography in Java (`code.google.com/archive/p/f5-steganography`), 2017.

[8] J. Fridrich and J. Kodovsky, Rich models for steganalysis of digital images, *IEEE Transactions on Information Forensics and Security*, vol. 7(3), pp. 868–882, 2012.

[9] B. Gruver, Smali Home (`github.com/JesusFreke/smali/wiki`), 2017.

[10] Guardian Project, Pixelknot: Hidden Messages (`guardianproject.info/apps/pixelknot`), 2017.

[11] Herodotus, *The Histories*, A. Burn (Ed.) and A. de Selincourt (Translator), Penguin Books, Harmondsworth, United Kingdom, 1954.

[12] F. Huang, J. Huang and Y. Shi, New channel selection rule for JPEG steganography, *IEEE Transactions on Information Forensics and Security*, vol. 7(4), pp. 1181–1191, 2012.

[13] JD Project, Java Decompiler – Yet Anther Fast Java Decompiler (`jd.benow.ca`), 2015.

[14] N. Johnson and S. Jajodia, Exploring steganography: Seeing the unseen, *IEEE Computer*, vol. 31(2), pp. 26–34, 1998.

[15] I. Lee and W. Tsai, A new approach to covert communications via PDF files, *Signal Processing*, vol. 90(2), pp. 557–565, 2010.

[16] S. Lyu and H. Farid, Steganalysis using higher-order image statistics, *IEEE Transactions on Information Forensics and Security*, vol. 1(1), pp. 111–119, 2006.

[17] W. Mazurczyk, P. Szaga and K. Szczypiorski, Using transcoding for hidden communications in IP telephony, *Multimedia Tools and Applications*, vol. 70(3), pp. 2139–2165, 2014.

[18] B. Pan, `dex2jar` (`github.com/pxb1988/dex2jar`), 2018.

[19] N. Provos, Outguess (`www.outguess.org`), 2017.

[20] N. Provos, StegDetect (`github.com/abeluck/stegdetect`), 2017.

[21] RADJAB, Da Vinci Secret Image (`play.google.com/store/apps/details?id=jubatus.android.davinci`), 2012.

[22] M. Sadek, A. Khalifa and M. Mostafa, Video steganography: A comprehensive review, *Multimedia Tools and Applications*, vol. 74(17), pp. 7063–7094, 2015.

[23] Sky Juice Software, Data Stash, Singapore (`www.skyjuicesoftware.com/software/ds_info.html`), 2017.

[24] C. Tumbleson and R. Winsniewski, Apktool: A Tool for Reverse Engineering Android APK Files, version 2.2.0, 2016.

[25] Twisted Pear Productions, Camouflage (`camouflage.unfiction.com`), 2018.

[26] D. Upham, Steganographic Algorithm Jsteg (`zooid.org/paul/crypto/jsteg`), 1993.

[27] A. Westfeld, F5 – A steganographic algorithm, in *Information Hiding*, I. Moskowitz (Ed.), Springer-Verlag, Berlin Heidelberg, Germany, pp. 289–302, 2001.

[28] WetStone Technologies, StegoHunt, Cortland, New York (www.wetstonetech.com/product/stegohunt), 2018.

[29] Wikibin, Jpegx (www.nerdlogic.org/jpegx/old/jpgx.html), 2017.