Tackling Android Stego Apps in the Wild

Wenhao Chen*, Li Lin*, Min Wu†, and Jennifer Newman*
* Iowa State University, Ames, IA, USA
E-mail: {wenhaoc, llin, jlnewman}@iastate.edu
† University of Maryland, College Park, Maryland, MD, USA
E-mail: minwu@umd.edu

***Abstract*—Digital image forensics is a young but maturing field, encompassing key areas such as camera identification, detection of forged images, and steganalysis. However, large gaps exist between academic results and applications used by practicing forensic analysts. To move academic discoveries closer to real-world implementations, it is important to use data that represent "in the wild" scenarios. For detection of stego images created from steganography apps, images generated from those apps are ideal to use. In this paper, we present our work to perform steg detection on images from mobile apps using two different approaches: "signature" detection, and machine learning methods. A principal challenge of the ML task is to create a great many of stego images from different apps with certain embedding rates. One of our main contributions is a procedure for generating a large image database by using Android emulators and reverse engineering techniques, the first time ever done. We develop algorithms and tools for signature detection on stego apps, and provide solutions to issues encountered when creating ML classifiers.**

I. Introduction

Digital image forensics is a term used in academia to describe the study of digital images for camera identification, image forgery, and steganalysis. With the popularity of mobile devices, camera identification and forgery detection are attracting research for more practical scenarios for digital image forensics. Important challenges remain, however, to detect steganography "in the wild", such as produced by mobile apps, and where academic research may provide impetus for tool development. In the penultimate case, evidence produced by a digital image forensic analysis for use in a court of law will be held to the Daubert standard [18], and assessed for scientifically-based reasoning and appropriate application to the facts.

Academic steganography and steganalysis techniques are very successful in the academic environment using sophisticated embedding and detection methods and data typically collected from digital still cameras [20, 7, 17, 8, 11, 12]. While mobile phones appear as part of criminal investigations on a regular basis, surprisingly, the authors found only one published research on steganography detection where a mobile phone app was used to produce the stego images [5]. This paper presents our results of the first in-depth investigation into detection of stego images produced by a number of apps on mobile phones.

In the academic community, a machine learning classifier is the first choice for steganalysis, and the algorithms are usually tested on a large database containing sufficient cover-stego pairs of images. However, as discussed in Section III, we show it is far from trivial to create appropriate training and testing data from mobile stego apps to use in machine learning (ML) classifiers. Moreover, unlike a published steganography algorithm, Android developers prefer using more sophisticated techniques, including a password for encryption or a secret preprocessing package, or coding techniques such as obfuscation. The third challenge is the variety of devices and input images for the apps, and ignoring the impact of the image source can cause unacceptable errors in detection of stego images.

As Albert Einstein once said, in the middle of difficulty lies opportunity. For data collection, we develop an Android camera app [4] that allows us to gather thousands of images by one device in just several hours. By using the most advanced tools from program analysis [14, 2, 21], we make great efforts to reverse-engineer many Android stego apps. In analyzing the code written by developers of stego app programs, we determine that, with few exceptions, most algorithms used to hide the message are far from the advanced algorithms published in academic research papers. For example, some of the app embedding algorithms were based on simple least significant bit (LSB) changes placed in an lexicographical order in the image. Some apps provide little security, even if a complicated embedding method was used, but strangely had a unique "signature" embedded, and make the stego image and its app easily identifiable as such. These are opportunities to take advantage of, and make steg detection easier.

Focusing on three phone models and six Android stego apps, we present answers to our fundamental question using signature-based detection and machine learning classifiers. While admittedly these initial experiments are limited in scope, the experiments are soundly designed and provide the first such deep study published using images from mobile phone apps. We note that there is a large potential for security risks if these apps continue to be used, but lack effective detection. Table II shows 6 apps from the Google Play store, with a minimum of 1000 to 100,000 downloads each, as of May 2018. Along with iPhone stego apps, as well as apps not posted including the recently identified "MuslimCrypt" [16], there is evidence that steganography continues to be a model of digital communication. Understanding how to detect stego images from stego apps has potential to help accurately assess the rate that stego images occur "in the wild."

The remaining sections of the paper are as follows. In Section II, we discuss the background of steganography, how it is manifested in mobile apps, the general workflow of stego apps, and the critical challenges in creating thousands of images from mobile apps to use in steganalysis. Section III describes the important process of creating the image dataset for all our experiments, including the non-trivial reverse-engineering procedure to generate the cover images corresponding to the app-generated stego images. In Section IV, we present the results of the signature-based detection of stego images from mobile apps, followed by results of the machine learning methods in Section V. Finally, in Section VI, we summarize our work and look ahead to future challenges.

II. Background

A. Steganography

Steganography offers techniques to send a payload without revealing its presence in the medium of communication. The focus of this paper is on digital images as the choice of medium for steganography. Unlike encryption of information, digital image steganography takes a payload, converts it to bits, and changes the bits of an image ever so slightly, so the changed image bits match the payload bits. This process is typically done in a way to avoid visual distortions of the image. In academic steganography, *cover image* is the term for an image in its form just prior to embedding by a stego algorithm, and the term *stego image* refers to an image output with hidden content. We call a *message* the bit string corresponding to the user-input content desired to be communicated. It does not include passwords or other information generated by the app code, such as length of the payload or bit locations, for example. The term *embed* is used to describe the algorithmic process of changing a cover image's bit values to represent the message bits. Typically, the change in bit values by the embedding algorithm results in change in color (or gray) intensity values by at most one. This leaves the overall visual content looking "normal" to human vision. We use the term *payload* to describe the combination of message, password, string length indicators and all other bit values that the app eventually embeds into the digital image. The *payload size* is the number of bits needed to represent the payload.

Many steganography algorithms exist to hide a payload in an image. One of the most common steganography algorithms uses the least significant bit values for embedding, and is adopted by many stego apps. Figure 1 provides an example of one simple LSB-replacement embedding for a grayscale image in the spatial domain, in which all the changes are highlighted by bold numbers. In the stego images, the LSB values of the cover images are replaced by the payload bits.

To make the payload more secure when embedding, a developer can follow additional steps. First, the payload bits themselves can be encrypted with a user-input, so that even if they are retrieved, the key is necessary for decrypting the payload. Second, the pixel locations where the payload bits are embedded can be selected in a random order, again using a key. As we will show in the following subsections, some stego apps use variations of the above methods to improve their level of security.

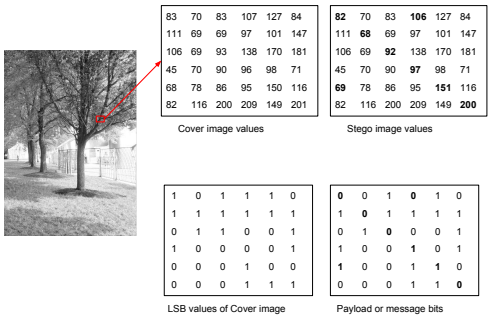

Fig. 1. An example of LSB embedding in the spatial domain.

B. General Workflow of the Embedding Process in Stego Apps

Although different stego apps may use different algorithms to create the stego images, they have many common features in their user interfaces. The user-input for stego apps include: (1) the input image, and (2) the message or file to be embedded, and optionally (3) the password. The output stego image is usually in PNG or JPEG format, depending on the image domain in which the payload is embedded. Figure 2 shows the general workflow of an embedding process in a stego app. Overall, an embedding process involves the following steps:

1) Decode domain values of input image;
2) Pre-process the domain values;
3) Pre-process the message;
4) Create and embed payload, and output stego image.

First, the user-input image is decoded into a bitmap of pixel values, and transformed into domain values. For spatial domain embedding, each domain value represents the RGB color and Alpha value of each pixel. For frequency domain embedding, the domain values represent the quantized DCT coefficients in the JPEG file. Additionally, the app may resize the cover image for different purposes such as reducing computational complexity, or increasing the cover image capacity.

Stego programs pre-process the message differently, such as encrypting the message before embedding to enhance security. Some stego programs attach signature strings or message length information so that the embedded message can be faithfully extracted by the receiver. A signature-based steg de-

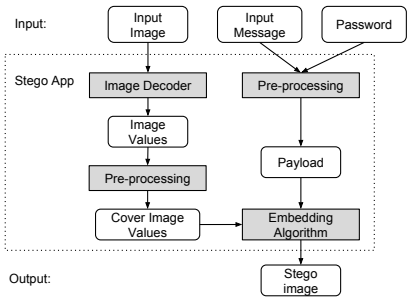

Fig. 2. General Workflow of an Embedding Process.

tection approach relies largely on the existence of the signature strings and length information. We provide more details on the signature-based detection approach in Section IV.

C. Steganalysis for Stego Images from Android Apps

Steganalysis has two main steps: first, to discover if hidden payload is contained within the image, and, if so, extract and decrypt the hidden message. The vast majority of papers in the academic community has so far focused on classifying an image as cover (innocent) or stego (with hidden content). Machine learning (ML) has proved to be a successful method in classifying cover-stego pairs in academic settings. A typical ML framework for steganalysis includes a labeled image database, a corresponding feature space to represent the images, and a classification algorithm to separate the stego images from the clean images. The performance of a ML algorithm can be evaluated by the average misclassification rate for targeted images with a certain *embedding rate*, where the *embedding rate* is defined as:

$$\frac{\text{\# of bits to represent the payload}}{\text{\# of the bits available to hide the payload (capacity)}}$$

One of the most popular image datasets in the academic community is BOSSbase [3], in which there are more than 10,000 cover images from seven digital still cameras. Using this dataset, steganalysts develop new algorithms to create stego images in a more secure way by sending the cover-stego pairs they create through advanced stego detectors. Most studies limit their cases to the balanced database scenario, in which the number of stego images is equal to the number of cover images, since under the assumption of balanced data, the average error rate of classification is sufficient to represent the performance of a steganalyzer. That is, if we let P_{MD} denote the percentage of misdetections and P_{FA} the percentage of false alarms, then for a dataset constituting 50% cover images and 50% stego images, the average error rate P_E for the detection is defined as:

$$P_E = \frac{1}{2}(P_{MD} + P_{FA}). \quad (1)$$

However, the scenario for detecting stego images created by mobile apps is different. First, unlike academic embedding algorithms or scripts which are capable of controlling the embedding rate by directly generating bit streams as payloads, a stego app encrypts real text together with a password (when available) into a bit stream as a payload for the target image. This means it is not trivial to generate a large amount of stego images at a specific embedding rate. Second, as we can see from Fig. 2, an input image provided by the user is processed before the embedding step in many of the stego apps, and therefore an input image is not necessarily the cover image in the traditional definition. Different stego apps apply different tools to process the input images into different image objects. In this paper, we view an image after the pre-processing procedure without any payload embedded as the *cover* image. In this way, a cover image is a clean image produced by the same processing libraries applied to its stego pair, and the cover-stego pairs will have the same visual property that human eyes cannot tell the difference. Extracting a cover image as an intermediate output from a stego app is another challenge, and the details of creating cover images by reverse engineering are presented in the next section.

III. Generating Stego Image Dataset

Since this is the first in-depth study on Android stego apps, a benchmarking database of images created by stego apps is essential. With real world stego apps, there are a few challenges we need to address when generating the dataset: (1) the absence of source code makes it non-trivial to analyze the apps; (2) in the scenario where the input image is transformed (e.g., downsampled) prior to embedding, the corresponding cover image is not saved by the app. In this section, we first describe the procedure of original image collection, in which we developed a camera app to achieve reliable and controlled image capture. We then explain our manual process of analyzing the stego app binaries to gain the ground truth of the apps' embedding process. Lastly, we explain the batch cover/stego image generation process that efficiently generates a large set of cover/stego image pairs using binary instrumentation.

A. Collection of original images

With the goal of studying Android apps, three mobile devices have been purchased: a Google Pixel, a Samsung Galaxy S7, and a OnePlus 5. To better understand and control the quality of images captured by smartphone cameras, we develop an Android app named "Cameraw" [4] to collect original images. Cameraw allows us to take a group of ten pairs of images consisting of the DNG format and the JPEG format at same time, for each fixed scene with various exposure parameters in one click. Table I summarizes the original images captured for this study. A total of 421 different scenes of JPEG and DNG images were collected across all three devices. These original images are used to generate stego images from the 6 stego apps, which we introduce next.

TABLE I
Summary of Original Images collected from 3 smartphones

Source Device	# Scenes	JPEG	DNG
OnePlus 5	120	1200	1200
Pixel1	187	1870	1870
Samsung S7	114	1140	1140
Overall	421	4210	4210

B. Generation of cover and stego images

1) Real-world Android Stego Apps: To generate the stego images and their corresponding covers, we have chosen 6 of the most popular stego apps from the Google Play Store, shown in Table II. We remark that the app *Steganography_M* [15] is actually named "Steganography" on Google Play Store. We append the letter M, which is the first letter of the author's name, to distinguish the app with the many other stego apps also named "Steganography" on Google Play Store. As shown in the "Output Format" column, 1 of

TABLE II
SIX SELECTED STEGO APPS FROM GOOGLE PLAY STORE

App Name	# Installs	Output Format	Open Source
PixelKnot	100,000 - 500,000	JPEG	yes
Steganography Master	10,000 - 50,000	PNG	no
Steganography_M	10,000 - 50,000	PNG	no
Da Vinci Secret Image	5,000 - 10,000	PNG	no
PocketStego	1,000 - 5,000	PNG	no
MobiStego	1,000 - 5,000	PNG	yes

the 6 apps produce stego images in JPEG format while the other 5 produce PNGs. The output format indicates the use of frequency domain embedding (JPEG) or spatial domain embedding (PNG). The "Open Source" column shows that only 2 apps have their source code publicly available, which makes the app analysis process non-trivial for the remaining four apps. We explain our process of reverse-engineering non-open source stego apps next.

2) Reverse engineering Android stego apps: For the 4 stego apps that are not open source, we utilize several Android program analysis tools to achieve reverse engineering. Given a stego app, we first use Apktool [21], a reverse engineering tool for Android, to decode the program binaries and resource files from the app's APK file. The program binaries are decoded into an intermediate code format called Smali [9]. The resources files are decoded into XML files that contain information about the app's graphical user interface (GUI). Next, we install and run the stego app on an Android device to test its user interface. We inspect the app's GUI structure while clicking through different screens, and use UIAutomator [1], an Android GUI testing tool, to retrieve the resource IDs for different GUI widgets. Using the resource IDs, we then identify the GUI widget (usually a button named "Embed") that initiates the embedding procedure, and locate the corresponding event handler method in the Smali code. After the core embedding algorithm code is located, we manually inspect the code to understand the key characteristics of the embedding algorithm.

Our goal of reverse engineering stego apps, is to study the following characteristics of an embedding algorithm, so that we may batch-generate image data for our experiments:

- **Embedding Domain**. The image domain in which the payload is embedded, either frequency domain or spatial domain.
- **Image Resizing**. Indicates resizing of the cover image prior to embedding. Stego apps can downsize the input image to reduce computation time, or upscale the input image to increase image capacity.
- **Payload Pre-processing**. The process that transforms the user input message into payload bits prior to embedding. For example, the input message can be encrypted, or appended with a signature string or length data.
- **Embedding Path**. The order in which the domain values are visited to embed the payload. Some apps use simple lexicographical embedding paths, while others use pseudo-random embedding paths with the user password as a seed.
- **Embedding Technique**. How the payload is embedded into the domain values. Common embedding techniques are LSB embedding in the spatial domain and F5 embedding [20] in the frequency domain. However, some of the apps we reverse-engineered have adopted their own unique embedding techniques.

Table III shows the embedding characteristics of the 6 stego apps we reverse engineered. Note that "user controlled" in the column "Image Resizing" means the app lets the user decide the output image size. In the sub-columns of "Payload Pre-processing," "yes" or "no" indicates the existence of payload processing steps.

The "Embedding Domain" column shows that *PixelKnot* embeds payload into the DCT coefficients of the frequency domain, while the others embed payload in the spatial domain.

The "Image Resizing" column shows that 2 out of 6 apps do not resize the cover image, while 3 apps may downsample the input image. The app *DaVinci Secret Image* allows the user to choose the image size from several options, including the option to maintain the original size.

The "Payload Pre-processing" column shows the three possible payload pre-processing options: encryption, signature string attachment, and length data attachment. Our investigation shows that prior to the embedding process, 3 apps perform encryption on the input message, 6 apps append signatures strings to the payload, and 3 apps append length data to the payload. While none of the apps perform all three payload pre-processing options, every app will attach at least either a signature string or a length information string to the payload. Such attachment to the payload is necessary for the app's extraction process, to identify the beginning and the end of the payload and correctly extract the message. However, it can leave patterns in the stego images that allow detection. We provide a detailed study on signature-based steg detection in Section IV.

The "Embedding Path" column shows that 3 apps embed the payload along pseudo-random paths, while the others use fixed embedding paths. The pseudo-random embedding path is generated by a pseudo-random number generator using the user input password as seed. The embedding path can be recreated using the same seed. The lexicographical embedding path starts from the top left of the image, and proceeds row by row or column by column sequentially. The app *MobiStego* uses a "regional lexicographical" order, where the cover image and payload are first split into multiple blocks, then a portion of payload bits is embedded into each block lexicographically.

The "Embedding Technique" column shows the variety of embedding techniques in the chosen apps. The frequency domain embedding app *PixelKnot* is based on the academic embedding algorithm F5. Out of the 5 spatial domain embedding apps, only *Steganography_M* and *PocketStego* use the standard LSB Replacement, while *DaVinci Secret Image* encodes the payload into alpha channel values. *MobiStego* embeds 6 bits of payload into a single pixel by replacing the two least significant bits of all three RGB channels, and *Steganography Master* embeds 8 bits of payload into one pixel by changing the decimal digit of each pixel's RGB value.

TABLE III
CHARACTERISTICS OF THE EMBEDDING PROCESS IN 6 STEGO APPS

App Name	Embedding Domain	Image Resizing	Payload Pre-processing			Embedding Path	Embedding Technique
			Encryption	Signature String	Length Data		
PixelKnot	frequency	downsampling	yes	no	yes	pseudo-random	F5
Steganography Master	spatial	no	no	yes	no	lexicographical	Base 10 LSD[a]
Steganography_M	spatial	no	no	yes	yes	pseudo-random	LSB
DaVinci Secret Image	spatial	user-controlled	no	yes	yes	lexicographical	Alpha channel encoding
PocketStego	spatial	downsampling	no	yes	no	lexicographical	LSB
MobiStego	spatial	downsampling	yes	yes	no	regional lexicographical	RGB channels LS2B

[a] Least Significant Digit replacement in base 10: replaces the "ones" digit in the cover image with one of the 3 base 10 digits of the message character, in each R-G-B planes.

3) Batch Image Generation through Instrumentation: To achieve the goal of batch-generating stego images with fixed embedding rates while saving the intermediate cover images, we use binary instrumentation to change the apps' binaries to generate cover/stego pairs. The binary instrumentation is achieved by first decoding the APK file into Smali code, then modifying the Smali code to add functionalities, and finally compiling the modified Smali code back to an APK file using *Apktool* [21]. Through instrumentation, we add two necessary functionalities to the stego apps: (1) saving the intermediate cover image along with its stego image pair, (2) automatically repeating the embedding process for all the input images.

Saving Intermediate Cover Images. As previously mentioned in Section II-C, statistical steganalysis benefits from having cover/stego pairs that went through the same image processing steps except for the embedding. However, real world stego apps, as shown in Table III, often preprocess the input image, which makes the ideal "cover" image unavailable. The implementation of the functionality of saving the covers varies and depends on the specific stego app. We achieved this in two ways: (1) modify the app's embedding function so that it accepts "empty payloads," in which case the produced stego image is equivalent to the cover image, or (2) add a new function to the stego app that can take the pre-embedding clean image data as input, and produce an image that has the same encoding or compression format as the stego image.

Batch Cover/Stego Generation. This functionality is necessary for processing the large amount of input images in our dataset in a timely manner. Each stego app has an added script module that recursively scans through folders of input images and calls the app's existing embedding functions to generate cover/stego pairs. Algorithm 1 shows the pseudo code of the batch cover/stego generation script. The script has 3 inputs: a set of clean images, a dictionary pool for the input message, and a set of target embedding rates. For each input image, the script first pre-processes the image and saves the intermediate cover (Lines 2-3). The image capacity is then calculated (Line 4). Lines 6-13 generate messages with different target embedding rates and create stegos. For each target embedding rate, we first calculate the length of the embedded payload *lp*, then calculate the length of the input message *lm* based on the knowledge acquired from reverse engineering the app. We then take a random segment of the dictionary with exact length *lm*, and proceed to call the app's embedding function. Relevant stego information and statistics, including the message, password, embedding rate, change rate, etc., are also stored along with the stego image in the database. Our large message dictionary contains text from 34 complete plays by Shakespeare.

Algorithm 1 Pseudo Code of Cover/Stego Generation Script

Input: input_images, dictionary, embedding_rates
1: **for each** $image$ in $input\_images$ **do**
2: $cover \leftarrow$ PREPROCESS($image$)
3: SAVEIMAGE($cover$)
4: $c \leftarrow$ CAPACITY($cover$)
5: **for each** $rate$ in $embedding\_rates$ **do**
6: $lp \leftarrow c \times rate$ ▷ embedded payload length
7: $lm \leftarrow$ CALCULATE(lp) ▷ input message length
8: $message \leftarrow$ GETMESSAGE($dictionary, lm$)
9: $password \leftarrow$ GETPASSWORD
10: $payload \leftarrow$ PREPROCESS($message, password$)
11: $stego \leftarrow$ EMBED($cover, payload$)
12: SAVEIMAGE($stego$)
13: SAVESTEGOINFO($message, password$)
14: **end for**
15: **end for**
16: **return**

Table IV shows the details of the stego images batch-generated from the 6 stego apps, using images in Table I as input. For the frequency domain app *PixelKnot*, we use the original JPEG and DNG images to generate stegos. For the other five spatial domain apps, we create PNG images that are center-cropped from the JPEG and DNG images, to reduce embedding time. For each input image, a total of 30 stego images are generated, including 5 stego images (with different embedding rates) from each of the 6 stego apps. The corresponding cover images for the stego images are also included in the dataset. In the next two sections, we present our study on signature-based steganalysis and machine learning steganalysis using the generated stego image dataset.

TABLE IV
SUMMARY OF STEGO IMAGES GENERATED FOR 6 STEGO APPS

Stego App	Input Format	Output Format	# Input	#Stegos	# Covers
PixelKnot	JPEG+DNG	JPEG	8420	42100	8420
Steganography Master	PNG	PNG	8420	42100	8420
Steganography_M	PNG	PNG	8420	42100	8420
DaVinci Secert Image	PNG	PNG	8420	42100	8420
PocketStego	PNG	PNG	8420	42100	8420
MobiStego	PNG	PNG	8420	42100	8420
Overall			50520	252600	50520

IV. Signature-based Steg Detection

This section provides a study on signature-based steg detection. We first discuss the definition of signatures, then analyze the signatures in the embedding process of 4 stego apps, and present the signature-based detection results on stego images from the 4 apps.

A. Embedding Signatures

In the context of this paper, a *signature* is a fixed pattern in the stego image that is unrelated to the cover image. In general, there are two types of embedding signatures: (1) fixed data written into fixed locations in the file headers, and (2) fixed data embedded into fixed locations in the image domain. As an example of signature type (1), the academic embedding algorithm F5 writes comment messages into the JPEG file header, and uses a specific bit in the JFIF header [10] to indicate the existence of user comments. Signature type (2) appears more frequently in our surveyed apps such as *DaVinci Secret Image* and *Steganography Master*, where a fixed string pattern is embedded into fixed pixel locations.

The main reason that such signatures exist is to provide auxiliary information for the extraction of payload. The extraction process requires auxiliary information, such as length data or fixed signature strings, to identify the beginning and ending of the embedded payload. Subsequently, embedding signatures can be used to identify stego images and even potentially extract the payload.

As previously mentioned in Section III-B2, we use reverse engineering techniques to analyze the embedding algorithms from stego apps. As shown in Table III, since the 2 apps *PixelKnot*, *Steganography_M* use pseudo-random embedding paths, their signature strings and length data are not in embedded in fixed locations. The other 4 apps *Steganography Master*, *DaVinci Secret Image*, *PocketStego*, and *MobiStego* each use a fixed lexicographical embedding path, which indicates a possible signature. Next, we analyze the embedding signatures of 4 stego apps and introduce our signature-based detection method.

B. Signature-based Detection Approach

The formats of embedded payload of the 4 stego apps are shown in Fig 3. Each app has a different way of converting the user input message and/or password into the embedded payload. Attached to the messages and passwords are two types of data: (1) signature strings which are represented with label "$", (2) length data, which is only used by *DaVinci*. The app *Steganography Master* joins the password and input message in plaintext and surrounds them with two pairs of fixed signature strings. The embedded payload in *DaVinci* consists of three segments: the signature string, the password in plaintext, and the message in plaintext. Each segment is also prefixed with a length data whose value is the number of bits of the segment. The app *MobiStego* first encrypts the input message using the password, then surrounds the encrypted message with a pair of signature strings. The app *PocketStego*

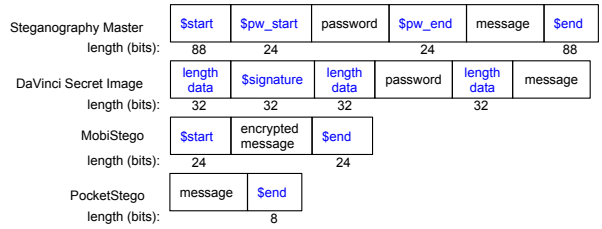

Fig. 3. Format of Processed Payload in 4 Stego Apps.

only appends a short 8-bit signature string at the end of the plaintext message.

To utilize the signature strings for steg detection, we also must know the apps' embedding paths and embedding techniques. The embedding path determines the pixel locations in which the signature string is embedded, and the embedding technique determines the insertion and extraction of bits into the pixel values. Given that all 4 stego apps use lexicographical embedding paths, and that 3 of them have fixed-length signature data at the start of the payload, we can identify the pixel locations that may contain unique signature data.

The embedding techniques of the 4 stego apps have been reverse engineered, as shown in Table III. *Steganography Master* first turns each 8 bits of payload into a decimal number ranging from 0 to 255. The 3 digits of the decimal number then replace the least significant base 10 digits of a pixel's R, G, B decimal values, respectively. *DaVinci* embeds 1 payload bit per pixel by setting the pixel's alpha value to 254 if the payload bit is 0, or to 255 if the payload bit is 1. *MobiStego* embeds 6 payload bits per pixel by replacing the least significant two bits of all three RGB values. *PocketStego* uses the standard LSB Replacement embedding where each payload bit overwrites a pixel's LSB in the Blue channel only.

With the knowledge of the signature strings, embedding paths, and embedding techniques from the 4 stego apps, we implement 4 stego detection functions. Each stego detection function corresponds to one of the stego apps. Each detection function takes a test image as input, and outputs a decision on whether the test image is a stego image produced by a specific stego app. The decision is made by extracting the embedded bits based on the stego app's embedding pattern, and checking whether the extracted payload matches the correct payload format. Next, we present our experimental results on signature-based steg detection.

C. Experimental Result

The image dataset for this experiment contains 202,080 images including 168,400 stego images and 33,680 cover images from the 4 stego apps: *Steganography Master*, *DaVinci Secret Image*, *MobiStego*, and *PocketStego*. The test results are shown in Table V. For each detector, the test data is grouped into two categories: (1) stego images generated from this stego app (labeled as *SM*, *DV*, *MS*, *PS* for short), and (2) all other images, including cover images and stego images from the other 3 stego apps. The detection results for the two groups of data are shown separately for each stego detector. As the

results show, the 3 stego detectors for *Steganography Master*, *DaVinci Secret Image*, and *MobiStego* correctly identify all stego images generated from their corresponding apps, while correctly distinguishing these stego images from cover images and other stego images. While the *PocketStego* detector has correctly identified all the *PocketStego* stego images, it also mis-identifies the majority of the other images.

TABLE V
RESULTS OF SIGNATURE-BASED STEG DETECTION

Stego App	Test Images	Image Count	Accuracy
Steganography Master	SM Stego Images	42,100	100%
	Other Images	159,980	100%
DaVinci Secret Image	DV Stego Images	42,100	100%
	Other Images	159,980	100%
MobiStego	MS Stego Images	42,100	100%
	Other Images	159,980	100%
PocketStego	PS Stego Images	42,100	100%
	Other Images	159,980	0.23%

The prefect results for *Steganography Master*, *DaVinci Secret Image*, and *MobiStego* detectors are as expected, as these apps have very distinctive signature strings in their payload. For example, as shown in Fig 3, *Steganography Master* has two fixed signature strings (112 bits in total), *DaVinci Secret Image* has 64 bits of distinct signature strings, and *MobiStego* has 24 bits of distinct signature strings, at the beginning of their payloads. On the other hand, *PocketStego* has only one 8-bit signature string at the end of the payload, without a fixed location. This "weak" signature can be found in not only the stego images from PocketStego, but also randomly occurs in 99.77% of other images as well, resulting in very poor accuracy.

Our test results demonstrate that it is possible to detect real world stego images based on app embedding signatures. The advantage of signature-based steg detection is that, by looking for known signatures in a test image, it can reliably identify stego images, and perhaps even extract the embedded payload. However, signature-based steg detection relies on the uniqueness of signature strings. If the developer changes the signature string with an update of the app, then it is possible the new signature will not be detected. A longer signature string with more patterns can provide better detection than shorter signatures. Furthermore, determination of the pattern of the signature string from the app binaries is not a trivial process, especially when the app has anti-analysis features such as obfuscation or native code. Our future work on signature-based detection is to automate the process of extracting signatures from stego apps using program analysis methods.

V. DETECTING STEGO IMAGES BY MACHINE LEARNING

Although we achieve near perfect results for detecting stego apps with distinctive signatures, many stego apps do not write any signature to the images they create. For those apps without signatures, we use machine learning methods. In this section, two Android apps, PixelKnot and Steganography_M, are selected for our studies, using two well-known methods: the F5 algorithm, and spatial LSB embedding. To our knowledge, this is the first time a ML detection algorithm is applied to identify stego images generated by mobile stego apps. We implement the CC-JRM [11] for feature extraction on JPEG images and SRM [6] for feature extraction on PNG images. The FLD ensemble classifer [12] performs the classification.

A. Case Study - PixelKnot

PixelKnot [19] is an Android app that uses a modified version of the embedding algorithm F5 [20] and it outputs a stego image in JPEG format. Before embedding, PixelKnot downsamples the input image if its size exceeds 1280*1280. The message is encrypted using one part of the password, while the other part of the password is used as seed for the F5 algorithm to generate a pseudo-random embedding path.

1) Experiments and Results: The goal of our first experiment is to study if the academic ML methods can be applied to detect the stego images created by PixelKnot. For every selected original image (DNG or JPEG), PixelKnot loads the standard Picasso package to downsample this larger image into a smaller bitmap object, which can be viewed as a 8-bit image in the spatial domain[1]. Cover and stego images with different embedding rates are generated from all original images by reverse engineering as discussed in the previous section. To evaluate the performance of the classic ML method, we implement CC-JRM and FLD ensemble classifiers for feature extraction and classification, respectively.

To give the first impression of ML detector for stego apps, for every device, we randomly select 850 original JPEG images as the input, and create the cover-stego pairs from those originals. We use 500 for training, and 350 for testing. The results are presented in Table VI.

TABLE VI
DETECTING STEGO IMAGES CREATED BY PIXELKNOT WITH 10% PAYLOAD (INPUT IMAGES ARE JPEG)

Data Source	Original Image Size	Avg. Error Rate
Google Pixel	4048 × 3036	**1.0%**
Samsung Galaxy S7	4032 × 3024	**7.6%**
OnePlus 5	4608 × 3456	**0.3%**
Mixture of above three devices	Flexible	**3.2%**

As we can see from the Table VI, when the image data sources are fixed, the results are quite encouraging. Even in the case when we mix the images from all three devices, the average error rate is just about 3%. Table VI shows that with the knowledge of the suspect image data source, for a fixed embedding rate, an academic ML based detection method works well in detecting the stego images generated from PixelKnot.

We point out that the results in Table VI are based on the assumption that we have knowledge of the source devices of the suspected images. However, this is not typical in a real-world scenario. The scenario when the source of the target image is not in the training database is called the *cover-source-mismatch problem* in steganalysis [13]. Table VII lists the

[1]We use grayscale images in our experiments.

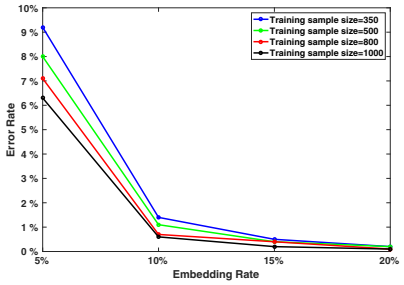

Fig. 4. Detecting stego images created by PixelKnot, where the original images are JPEG images collected from Google Pixel.

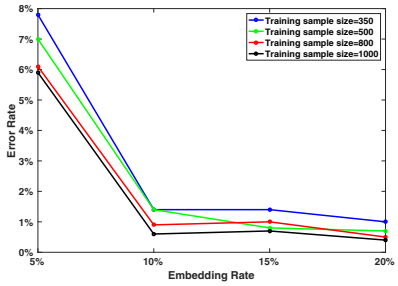

Fig. 5. Detecting stego images created by PixelKnot, where the original images are DNG images collected from Google Pixel.

results in the case when the sources of test images are not involved in the training database.

TABLE VII
DETECTING STEGO IMAGES CREATED BY PIXELKNOT WITH 10% PAYLOAD, IN THE COVER-SOURCE-MISMATCH CASE.

Test Data Source	Training Data Source	Avg. Error Rate
Google Pixel	Samsung Galaxy s7 & OnePlus 5	**1.3%**
Samsung Galaxy s7	Google Pixel & OnePlus 5	**40.0%**
OnePlus 5	Google Pixel & Samsung Galaxy s7	**35.7%**

In Table VII, the average error rates are not at the same level for the three cover-source-mismatch cases. Although the error in testing images from Google Pixel is almost as low as the error in Table VI, the error rates of detecting stego images from the other two devices when they are out of the training datasets, are much greater than those in Table VI. Our preliminary results show that it is not desirable to use only one or two devices to build a classifier for blind detection on multiple devices. Thus, knowing the source of the target images in detecting stego apps will significantly reduce the error rate for a ML based analyzer.

TABLE VIII
AVERAGE ERROR RATES FOR DETECTING STEGO IMAGES CREATED BY PIXELKNOT WITH DIFFERENT PAYLOAD SIZES.

Training Set: / **Test Set:**	**5% Stego**	**10% Stego**	**15 % Stego**	**20% Stego**
5% Stego	7.9%	41.4%	47.9%	49.6%
10% Stego	4.7%	1.0%	12.9%	39.3%
15% Stego	3.9%	0.9%	0.4%	2.86%
20% Stego	4.4%	0.7%	0.4%	0.2%

In the previous experiment, the embedding rate is fixed at 10% for all stego images we created. To study the possibility of detecting stego images that have different embedding rates, we use the previous input images collected from Google Pixel to generate three more sets of stego images with three different embedding rates: 5%, 15% and 20%. For each subset, we build a stego classifier for a fixed embedding rate. The results are presented in Table VIII.

In Table VIII, for the same test data, the lowest error rate always occurs in the diagonal entries, for which the training stego images have the same embedding rate as the images for testing. Another interesting phenomenon is that the many error values located above the diagonal are extremely high, while error values below the diagonal look acceptable. Table VIII gives us an impression that it may be possible to apply well-trained stego classifiers based on 5% embedding rate to suspect images having unknown payloads. We have to admit that this conclusion is limited to the case when all images are from only one mobile phone, which is the Google Pixel, and it is left to future research for the verification by larger-scale experiments for a variety of sources.

The performance of a ML based steg detector can depend on the training sample size and the embedding rate for the target data. To test this, with the Google Pixel data, we use four different sample sizes for the training set, at four different embedding rates, and average the error rate over 10 different random drawings of the training data. The results are shown in Fig. 4 and Fig. 5. It is clear from both figures that as the embedding rate increases, the average error rate decreases, for all sample sizes. Also, increasing the size of the training set slightly reduces the error, especially when the embedding rate is very low. Futhermore, because PixelKnot preprocesses and downsamples the input images, the choice for the format of input image, JPEG or DNG, does not cause a significant difference between the results for the two experiments.

B. Case Study - Steganography_M

To study how well a ML detection method works in the case when stego images are created by spatial domain embedding algorithms, we run our second case study for the app Steganography_M [15]. Steganography_M is an Android stego app that uses spatial domain embedding with pseudo-random embedding paths and implements an embedding algorithm very similar to the standard LSB spatial embedding. From the clues we found during the program analysis, we know that the cover image is not resized before embedding, and the pseudo-random embedding path is generated from the user password.

1) Experiments and Results: Since there is no preprocessing of the image data by Steganography_M, the noise levels of input images can affect the detection results significantly. For that reason, two original image formats, JPEG and DNG, are used in this case study. However, with limited time and resources, we use cropped grayscale PNG images with dimension 512 $\times$ 512 from original images as the input images. One benefit of such a choice is that feature extraction for smaller images is much more efficient. Another important

reason is that using processed PNG images for the input can be viewed as clean cover images, since they are not compressed or downsampled by the app. With the help of Google Android emulator, we create thousands of stego images with varied embedding rates for all original images, both JPEG and DNG, collected by the app Cameraw installed on all three devices.

In this experiment, SRM and ensemble classifiers are applied for feature extraction and classification, respectively. For each fixed embedding rate, we randomly select a sample of images for training and then test the classifier on another sample. Table IX provides the result when JPEG images are used to generate the inputs. As we can see from Table IX, when original images are in JPEG format, the ML based steg detectors work so well that even for a very low embedding rate ($< 5\%$), the error rates are never above 3% for all devices.

TABLE IX
DETECTING STEGO IMAGES CREATED BY STEGANOGRAPHY_M .ORIGINAL IMAGES ARE JPEG, TRAINING SAMPLE SIZE = 500, TEST SAMPLE SIZE = 350.

Data Source	Embedding Rate	Avg. Error Rate
Google Pixel	3%	**0.4%**
	5%	**0.0%**
	8%	**0.0%**
Samsung Galaxy S7	3%	**1.3%**
	5%	**0.9%**
	8%	**0.4%**
OnePlus 5	3%	**2.7%**
	5%	**1.4%**
	8%	**1.0%**

As we did for Pixelknot, the results of training on one embedding rate and detecting with different embedding rates is summarized in Table X. In this experiment, only JPEG images from Google Pixel are used to generate input images, and for every classifier, 500 random pairs of cover-stego images are used for training, 350 for testing. The results in Table X are very similar to what we concluded for the app Pixelknot.

TABLE X
AVERAGE ERROR RATES FOR DETECTING STEGO IMAGES CREATED BY STEGANOGRAPHY_M WITH DIFFERENT PAYLOAD SIZES.

Training Set: Test Set:	**2% Stego**	**3% Stego**	**5 % Stego**	**7% Stego**
2% Stego	2.9%	28.4%	48.0%	49.6%
3% Stego	2.4%	0.4%	0.6%	28.4%
5% Stego	2.7%	0.6%	0.0%	0.1%
7% Stego	2.6%	0.4%	0.0%	0.0%

However, as we mentioned above, there is no preprocessing procedure for the input images by *Steganography_M*. As a result, compared to using JPEG as the data source, the noisy DNG images significantly increases the detection errors. To illustrate this phenomenon, using the device Google Pixel, we randomly select 350 JPEG images and 350 DNG images to create cover-stego pairs as the test dataset at different embedding rates, repeat ten times, and for each time and each embedding rate, we create the corrsponding training datasets with four different sample sizes. The results, as shown in Fig. 6and Fig. 7, show that using the DNG images to generate the input images for *Steganography_M* produces higher error

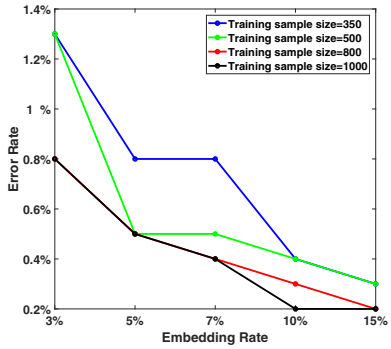

Fig. 6. Detecting stego images created by Steganography_M, where the original images are JPEG images collected from Google Pixel.

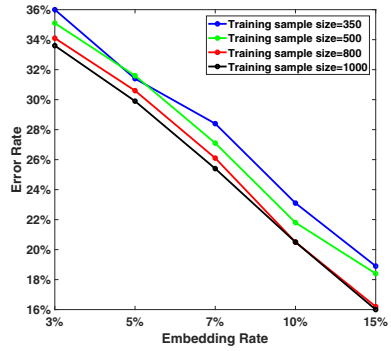

Fig. 7. Detecting stego images created by Steganography_M, where the original images are DNG images collected from Google Pixel.

from a ML classifier. Moreover, even with 1000 cover-stego pairs for training and a high embedding rate of 15%, the error of misclassification is at a minimum of 16% for DNG images from one phone.

Our experiments show that the knowledge of stego apps' embedding algorithm can be beneficial for stego detection. In both experiments, we assume that we have the knowledge of the app that created stego images. But this is not common in practice, as we may have very little information about which stego apps were used. To improve the feasibility of machine learning based detection, our future work includes: (1) analyzing and studying more real world stego apps, and (2) exploring the possibility of training with one known app and detecting stego images from a different app.

VI. CONCLUSION

In this paper, we analyze six Android apps that implement steganography algorithms. A major contribution of this paper is our procedure to analyze the code in the app by applying reverse-engineering techniques to the binary code. We use instrumentation techniques to perform the non-trivial task to batch-generate cover-stego image pairs for machine learning steganalysis. Thus, with appropriate numbers of images, we create machine learning classifiers and perform successful

steganalysis on stego images created from two mobile apps. We also present a detailed analysis of four stego apps that contain a signature, and perform steg detection on these images. While this is currently done in a manual process, our future efforts will investigate methods to automate the program analysis of app code. Another challenge is to go beyond the task of identifying stego or innocent images: the extraction of hidden contents. A program analysis approach can also be useful to solve this problem. Another future challenge we plan to implement are other learning paradigms, including deep learning.

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.

References

[1] UI Automator. https://developer.android.com/training/testing/ui-automator.html, 2017. Last Accessed: 2017-12-20.

[2] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[3] P. Bas, T. Filler, and T. Pevnỳ. Break Our Steganographic System: The Ins and Outs of Organizing BOSS. In *Information Hiding*, pages 59–70. Springer, 2011.

[4] W. Chen. Cameraw, an Android camera app for digital image forensics. Technical report, CSAFE, Iowa State University, Oct. 2017.

[5] W. Chen, Y. Wang, Y. Guan, J. Newman, L. Lin, and S. Reinders. Forensic analysis of steganography apps on android. In *IFIP International Conference on Digital Forensics*. Springer, 2018.

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

[7] J. Fridrich, T. Pevný, and J. Kodovský. Statistically Undetectable Jpeg Steganography: Dead Ends Challenges, and Opportunities. In *Proceedings of the 9th Workshop on Multimedia & Security*, MM&Sec '07, pages 3–14, New York, NY, USA, 2007. ACM.

[8] H. Gou, A. Swaminathan, and M. Wu. Noise features for image tampering detection and steganalysis. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, volume 6, pages VI–97. IEEE, 2007.

[9] B. Gruver. Smali. https://github.com/JesusFreke/smali/wiki. Accessed: 2017-09-17.

[10] E. Hamilton. Jpeg file interchange format. 2004.

[11] J. Kodovskỳ and J. Fridrich. Steganalysis of jpeg images using rich models. In *Media Watermarking, Security, and Forensics 2012*, volume 8303, page 83030A. International Society for Optics and Photonics, 2012.

[12] J. Kodovsky, J. Fridrich, and V. Holub. Ensemble classifiers for steganalysis of digital media. *IEEE Transactions on Information Forensics and Security*, 7(2):432–444, 2012.

[13] J. Kodovskỳ, V. Sedighi, and J. J. Fridrich. Study of cover source mismatch in steganalysis and ways to mitigate its impact. In *Media Watermarking, Security, and Forensics*, page 90280J, 2014.

[14] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[15] J. Mexnik. Steganography. https://play.google.com/store/apps/details?id=com.meznik.Steganography, 2014. Last Accessed: 2017-06-10.

[16] L. Motherboard, by Vice Media. Muslimcrypt. https://motherboard.vice.com/en_us/article/ne4x7w/muslim-crypt-jihadi-encryption-app, 2018. Last Accessed: 2018-01-11.

[17] T. Pevny, P. Bas, and J. Fridrich. Steganalysis by subtractive pixel adjacency matrix. *IEEE Transactions on Information Forensics and Security*, 5(2):215–224, June 2010.

[18] H. S. Stern. Statistical issues in forensic science. *Annual Review of Statistics and Its Application*, 4:225–244, 2017.

[19] The Guandian Project. Pixelknot. https://play.google.com/store/apps/details?id=info.guardianproject.pixelknot, 2017.

[20] A. Westfeld. F5—a steganographic algorithm. In I. S. Moskowitz, editor, *Information Hiding*, pages 289–302, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[21] R. Winiewski and C. Tumbleson. Apktool - A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/, 2017.