Stegomalware: Playing Hide and Seek with Malicious Components in Smartphone Apps

Guillermo Suarez-Tangil, Juan E. Tapiador, and Pedro Peris-Lopez

Department of Computer Science, Universidad Carlos III de Madrid
Avda. Universidad 30, 28911, Leganes, Madrid, Spain
guillermo.suarez.tangil@uc3m.es, jestevez@inf.uc3m.es, pperis@inf.uc3m.es

Abstract. We discuss a class of smartphone malware that uses steganographic techniques to hide malicious executable components within their assets, such as documents, databases, or multimedia files. In contrast with existing obfuscation techniques, many existing information hiding algorithms are demonstrably secure, which would make such *stegomalware* virtually undetectable by static analysis techniques. We introduce various types of stegomalware attending to the location of the hidden payload and the components required to extract it. We demonstrate its feasibility with a prototype implementation of a stegomalware app that has remained undetected in Google Play so far. We also address the question of whether steganographic capabilities are already being used for malicious purposes. To do this, we introduce a detection system for stegomalware and use it to analyze around 55K apps retrieved from both malware sources and alternative app markets. Our preliminary results are not conclusive, but reveal that many apps do incorporate steganographic code and that there is a substantial amount of hidden content embedded in app assets.

Keywords: smartphone security, malware, steganography, obfuscation

1 Introduction

Malware for smartphones has rocketed over the last few years. Such a phenomenon is intimately related to the popularity of smartphone platforms and the substantial rise in the number of apps available for download in online markets. While these two facts have contributed to create new business models and reshape the way we communicate, malware writers have taken advantage of the possibilities offered by smartphones for spying on the user's activities, stealing his identity, or committing fraud, among other malicious activities [30].

Thwarting malware attacks in smartphones is still a formidable challenge. On the one hand, battery-powered smartphones do not possess enough computing capabilities to constantly check for attempts of executing malicious operations. Furthermore, distinguishing what is malware from what is not is far from being easy. In any case, the most common distribution strategy for smartphone malware is still the use of both official and unofficial markets [25]. Attackers

simply upload malicious apps to the market, sometimes using a stolen identity, and users get infected by just downloading and installing the app.

In the case of official markets, operators are generally concerned about the security of the software they distribute. To address malware attacks, most markets implement a revision process that presumably includes various security checkings [15]. Malware writers are constantly seeking ways of evading detection. For instance, in the so-called update attacks, the app just contains a "hook" that, once installed in the user's device, downloads and executes a malicious payload from a external server pretending to be a required update of the app.

Smartphone malware is becoming increasingly stealthy and recent specimens are relying on advanced code obfuscation techniques to evade detection by security analysts [24]. For example, *DroidKungFu* has been one of the major Android malware outbreaks. It started on June 2011 and has already spanned over at least six variants. *DroidKungFu* has been mostly distributed through official or alternative markets by piggybacking the malicious payload into a variety of legitimate applications. The malicious payload with *DroidKungFu*'s actual capabilities is encrypted into the app's assets folder and decrypted at runtime using a key placed within a local variable belonging to a specific class module. *GingerMaster* is another representative example of smartphone malware that deliberately tries to hide itself. In this case, the main payload is stored as PNG and JPEG pictures in the asset folder. Such files are loaded as regular pictures by a special hook within the app and then interpreted as code.

Examples such as the two described above do not abound yet but are becoming increasingly common. In the case of traditional platforms such as PCs, malware writers have made use of obfuscation techniques for decades, ranging from simple packing algorithms to using more sophisticated polymorphic and metamorphic engines (see [16] for an excellent overview). Such techniques transform malicious code to make it difficult to understand, analyze, and detect by static analysis.

1.1 Contributions

Motivated by the increasingly creative ways used by smartphone malware to hide malicious components and evade detection, we pose the question of how information hiding techniques could be used by malware writers to achieve the same purpose. Contrarily to the ad hoc—and, in many cases, sloppy—measures seen today to obfuscate malicious components, modern information hiding techniques could provide a simple yet theoretically robust way of hiding executable pieces in assets such as pictures, databases, multimedia files, etc. To the best of our knowledge, this is the first paper that looks into this issue.

In summary, in this paper we make the following contributions:

1. We describe a class of smartphone malware that uses steganographic techniques to hide malicious executable components within an app's assets. We call this "stegomalware" and argue that steganographic algorithms provide

malware writers with a mechanism for hiding malicious payloads more secure than obfuscation techniques currently in use.
2. We discuss various architectures for stegomalware depending on the location of the asset with hidden capabilities and the algorithm required to extract it.
3. We show that current app markets may be vulnerable to stegomalware. In particular, we describe a prototype implementation of a stegomalware sample for Android platforms that is available for download in Google Play and has remained undetected so far.
4. We introduce a detection system for stegomalware that combines steganalysis techniques with the detection of steganographic algorithms in the app code.
5. Using an implementation of our stegomalware detection system, we address the question of whether steganographic capabilities are being already used for malicious purposes. We analyze around 55K apps retrieved from both malware sources and alternative app markets. Even though our preliminary results are not conclusive, we found that many apps do incorporate steganographic code and that there is a substantial amount of hidden content embedded in app assets.

2 Information Hiding Techniques and Related Work

In this section, we provide a brief background on information hiding techniques, including some formal definitions, common stegosystems, and techniques to detect the presence of hidden information. Because of their popularity, we will focus our discussion on stegosystems that hide information in pictures. We will revisit this point later and discuss alternative stegosystems for the type of malware discussed in this paper. Interested readers can find good introductions to this discipline in [13, 17, 19]. Finally, we briefly review the literature related to malware in smartphones.

2.1 Stegosystems

Modern steganography studies techniques to hide the presence of information by embedding secret messages within other, seemingly harmless digital objects. According to the standard terminology of information hiding [18], the original object used to hide information is called the *covertext*, whereas the same object after embedding the secret message is called the *stegotext*. The embedding process depends on a key and the adversary (known as the *warden*) is generally assumed to know everything but the key.

Formally, a symmetric[1] stegosystem is a triple of probabilistic polynomial-time algorithms $(\mathsf{SK}, \mathsf{SE}, \mathsf{SD})$ with the following properties [3]:

- The key generation algorithm SK takes a security parameter n as input and returns a stegokey sk.

[1] This definition can be naturally extended to public-key stegosystems [3].

– The steganographic encoding algorithm SE takes as input the security parameter n, the stegokey sk, and a message $m \in \{0,1\}^l$ and outputs a stegotext c belonging to the covertext space C. The algorithm may access the distribution of C if needed.
– The steganographic decoding algorithm SD takes as input the security parameter n, the stegokey sk, and an element $c \in C$, and outputs either a message $m \in \{0,1\}^l$ or a special symbol $\perp$ denoting that no message is embedded in c.

A stegosystem must be *reliable*, i.e., the probability that

$$\mathsf{SD}(1^n, sk, \mathsf{SE}(1^n, sk, m)) \neq m \tag{1}$$

must be negligible in n for all messages m and all stegokeys sk. Informally speaking, the *security* of a stegosystem is related to the probability that an adversary detects the presence of an embedded message. Thus, a secure stegosystem is one in which covertexts and stegotexts are indistinguishable. This notion can be formally established as a distinguishability experiment similar to those common in the field of provable security. In turn, this gives rise to various notions for secure stegosystems, including perfect security, statistical security, and computational security (see [3] for further details).

Apart from their security, there are other aspects of practical relevance for stegosystems. One is their *capacity*, defined as the maximum number of secret bits that can be securely embedded. In most practical stegosystems there is a trade-off between security and capacity, so the longer the embedded message, the more distinguishable the stegotext becomes. Another relevant property is the *robustness* of the stegosystem. Informally speaking, robustness measures the amount of distortions that a stegotext can endure until recovering the embedded message becomes impossible. This is a key property for applications such as watermarking and fingerprinting, where the focus may not be on hiding the presence of some embedded mark, but on preventing an adversary from removing it without degrading the quality of the data object.

2.2 Common Steganographic Algorithms

A variety of stegosystems have been proposed for embedding data in all sorts of digital sources, including text file formats, compiled code, images, audio, and video. There are, in fact, few digital formats where some opportunity for steganography has not been identified. Stegosystems for multimedia objects—and, in particular, for images—are among the most popular schemes because of the high embedding capacity offered by digital pictures and the proliferation of images in Internet. Thus, most papers in this field have concentrated on JPEG images, although the underlying ideas and algorithms are generally applicable to other formats as well.

The JPEG image format is based on taking the discrete cosine transform (DCT) of 8x8-pixel blocks of the image, producing 64 DCT coefficients. Once

quantized, the least-significant bits (LSB) of each coefficient are modified to embed hidden messages. Note that the modification of a single DCT coefficient affects all 64 pixels in the block. We next describe four popular JPEG stegosystems that use some form of LSB embedding in the frequency domain (see [4] for a recent survey).

`Jsteg` Proposed by Derek Upham, this is one of the earliest stegosystems for JPEG images [27]. `Jsteg` replaces the LSB of the DCT coefficients by the secret message bits, skipping those coefficients with the values 0 or 1. The image is scanned sequentially and the algorithm does not support random bit selection. The key, if any, is used to encrypt the message before embedding. `Jsteg-shell` is a popular Windows front-end for Jsteg that encrypts the message with RC4.

`JPHide` This is a stegosystem proposed by Allan Latham that supports compression of the secret message and encryption with Blowfish. The algorithm is also based on replacing the LSB of the DCT coefficients but does not do it sequentially. Instead, it uses a fixed table to determine which coefficient will be changed next. Furthermore, a pseudorandom number generator (PRNG) is used to skip some of them, where the probability of skipping changes depending on how many bits have been embedded already and how many are left. `JPHide` can also use the second LSB in some cases.

`OutGuess` This is yet another JPEG stegosystem using LSB encoding in the DCT coefficients. Contrarily to the two previous algorithms, `OutGuess` chooses the coefficients randomly using a PRNG initialized with a user-provided password. The content is also encrypted using RC4 with the same password used for the PRNG.

`F5` Developed by Andreas Westfeld, `F5` can be seen as an evolution of the stegosystems described above. It introduces a number of novel ideas, including the use of a matrix encoding to reduce the number of necessary changes and a permutative straddling to uniformly spread out the modifications over the whole covertext. `F5` reduces the propagation of steganographic information over the carrier medium. This feature makes `F5` robust against certain distortions such as resizes or rotations. Full details are available in [29].

2.3 Steganalysis

Steganographic encoding algorithms leave traces on the stegotexts as a consequence of the alterations required to embed the message. Such traces are instrumental in facilitating detection, i.e., distinguishing whether an object has or has not embedded information. This is the main goal of a *passive warden*, i.e., an adversary who can read objects and must determine if a secret communication is taking place. Note that a correct detection defeats the main purpose of steganography, which is hiding the very presence of a communication. In general, exposing the content of such a secret communication is another problem entirely.

Contrarily to passive wardens, an *active warden* is not concerned with detecting secret communications, but with destroying them. Active warden techniques

introduce deliberate modifications in all digital objects in the hope that any potential hidden content would be rendered unusable. In some domains, this is just too costly and some form of sampling must be performed.

In what follows, by *steganalysis* we will refer to the process of distinguishing whether an object has or has not hidden information. A steganalytic technique is often presented in the form of *distinguisher*, this being a test that returns some measure of the likelihood of the input sample having embedded information. Steganalytic techniques can be classified according to various criteria. A *targeted* distinguisher focuses on the artifacts produced by one specific algorithm and, therefore, can only detect if that algorithm has been used. Contrarily, a *blind* (or *universal*) distinguisher identifies statistical alterations caused by any steganographic encoding algorithm [7]. An example of a blind steganalysis is the χ^2-attack, based on applying a χ^2 test to compare the distributions of adjacent DCT coefficients, which is similar in images with hidden data embedded [20]. Fridrich et al. provide in [9, 10] an overview of feature-based steganalysis for JPEG images and its implications for future designs of stegosystems.

There are a number of freely available implementations of the main steganalytic techniques proposed so far, including:

`Stegdetect` [20] is a popular and free steganalytic tool. It includes a number of distinguishers to detect the presence of hidden data in images and is able to identify the method used during embedding process. `Stegdetect` is the *de facto* tool used by security and forensic practitioners due to its excellent capabilities and its free and open nature. A recent study by Khalind et al. [14] has revisited its features and warned about the implications of its false positive ratio.

`VSL` **(Virtual Steganographic Laboratory)** [8] is a suite of steganalytic techniques that includes some of the most popular techniques, including RS-Analysis (a distinguishing algorithm for LSB methods) and a blind steganalysis technique based on Support Vector Machines.

`SSS` **(Simple Steganalysis Suite)** [2] is another publicly available implementation of various image steganalytic techniques, including χ^2-attack and various histogram-based tests.

2.4 Thwarting Malware in Smartphones

A substantial amount of recent work has addressed the problem of analyzing malware in smartphones using a variety of techniques [6, 25]. Static analysis techniques are well known in traditional malware detection and have recently gained popularity as efficient mechanisms for market protection [26, 1]. However, current static techniques fail to identify malicious components when they are obfuscated or embedded separately from the code (e.g., hidden into an image) [22, 12]. Approaches based on dynamic code analysis [11] are promising, but current works [6, 21, 23] only provide an holistic understanding of the behavior of an app. This feature challenges the identification of malware using steganography.

More recent approaches focus on detecting hidden functionality [24] within components of an app. Although this technique has shown to be promising, it requires a non-negligible overhead derived from the dynamic execution of every app analyzed. As regards the various ways to hide or locate hidden code in apps using steganography and steganalysis, to the best of our knowledge this is the first work addressing this issue in smartphones.

3 Stegomalware

This section introduces the idea of an app that uses a stegosystem to hide a malicious component within its assets and then extracts and executes it dynamically. We then discuss various architectures for such a stegomalware and describe a prototype implementation for Android platforms that is available for download in Google Play and has remained undetected so far.

3.1 Hiding Malicious Code in App Assets

Malware developers can use steganographic capabilities to hide malicious components within an app resources. Such resources depend on the particular app and may include images, audio, video, databases, and text files in a variety of formats (e.g., plain text, XML, and HTML). Practical stegosystems for all these digital objects have been proposed, some of them with a reasonable security level. Moreover, there is a variety of freely available implementations of such stegosystems that are exceedingly simple to use within an app.

The main goal pursued by an attacker who uses a stegosystem to securely embed a piece of malicious code into an app resource is to evade detection, particularly static analysis based approaches implemented by market operators. Hiding malicious components may also difficult malware analysis, as the payload will be located in places that security analysts could overlook. More importantly, the piece of malicious code is not accessible to analysis. This is a key difference—and a substantial advantage for malware writers—with respect to traditional malware obfuscation techniques: after obfuscation, malicious code may be difficult to recognize, but it is still somewhere in the app. In contrast, in a stegomalware specimen the malicious component is revealed at execution time only. Thus, it will not match any signature even if the search includes the asset where it is hidden.

A stegomalware contains the following three basic components:

- *A stegotext* R, this being one of the app assets. R is the result of embedding a malicious payload p with a steganographic encoding algorithm SE using some stegokey.
- *A stegokey* sk required to extract p from R. In case of using a symmetric stegosystem, sk is the stegokey that was used to embed p in R.
- *A steganographic decoding algorithm* SD needed to recover p from R using sk.

Table 1: Three variants of stegomalware and their activation procedures.

Type	Locally Available	Remotely Available	Activation
Type 0	R, sk, and SD	nothing	1. Get sk and R from the app resources 2. Recover the payload: $p = \mathsf{SD}(sk, R)$. 3. Execute p.
Type I	l and SD	R and sk	1. Get URL l from the app resources. 2. Connect to l and retrieve (R, sk). 3. Recover the payload: $p = \mathsf{SD}(sk, R)$. 4. Execute p.
Type II	l and SD	R and sk	1. Get URL l from the app resources. 2. Connect to l and retrieve (R, sk, SD). 3. Recover the payload: $p = \mathsf{SD}(sk, R)$. 4. Execute p.

These three elements (R, sk, SD) can be packaged together and distributed with the app, or dynamically retrieved at runtime from an external server. Based on this, we next describe three architectural choices for stegomalware. This list does not intend to be exhaustive and more complex variants are possible.

3.2 Type 0: Autonomous Stegomalware

One simple choice is to have all the stego material (R, sk, SD) distributed with the app. The asset R is the less problematic of the three, as it can just be put in the asset folder with the remaining resources. The algorithm SD must be part of the code assets so the app can invoke it to retrieve p. The stegokey sk must be part of the app too, either hardcoded as a variable somewhere in the code, or else distributed in some other asset (e.g., a text file, a database, etc). Note that sk is not necessarily a random string, as many stegosystems accept alphanumeric passwords from which subsequent keying material is derived.

This type of stegomalware is fully autonomous and does not depend on a remote infrastructure to achieve its goals (see Table 1). This, however, comes at a price: the presence of R, sk, and SD may facilitate detection. We will discuss this issue in more detail later in Section 4.

3.3 Type I: Stegoupdate Attacks

A more flexible alternative to Type 0 stegomalware consists of retrieving the stegotext R remotely during activation. This allows for using different malicious payloads depending on the attacker's goals, the particular target, etc. The app must necessarily contain the decoding algorithm SD and possibly sk, although it is generally more convenient to have sk associated with the particular R and, therefore, also dynamically downloaded. This variant introduces the need for incorporating into the app the location (e.g., an URL) of the external server

from where (R, sk) will be fetched. The activation procedure is quite simple (see Table 1) and involves fetching R and sk, extracting p locally, and then executing it.

This type of stegomalware can be seen as a variant of the classical update attacks [30] in which the malicious payload is downloaded embedded into an innocuous-looking object. This would evade detection schemes based on monitoring update traffic and preventing the execution of downloaded code.

3.4 Type II: Agnostic Stegomalware

In a more radical setting, the app could be totally agnostic of the stegosystem used to embed the payload. Thus, after activation the app would connect to a remote site and download R, sk, and SD. The decoding algorithm would be then used to extract the malicious payload from R.

The key idea here is not to distribute SD within the app, so even a detailed code analysis would not raise any suspicion. This idea admits some minor variations. For example, R and sk might actually be part of the app, so the update engine just downloads SD and uses it to extract the malicious payload. Similarly, in a collusion scenario, the app could pass R and sk on to another app in the same device that implements SD. This second app would first extract the payload and pass it back to the original app (e.g., by putting it in shared space) or just execute it.

3.5 A Proof-of-Concept Implementation

We implemented a prototype of a simple autonomous stegomalware (Type 0) for Android platforms. The app is named `LikeImage` and contains a malicious payload embedded into an image that is part of the app resources (see Fig. 1). In execution time, the payload is extracted from the image and executed dynamically. We used an open source Java implementation[2] of `F5` for JPEG images as the underlying stegosystem. Fig. 2 shows the original image and the one that is actually distributed with the app after embedding the payload. As expected, both images look exactly the same.

The malicious payload is a Java JAR library compiled in Dalvik Executable (DEX) Format. Fig. 3 shows a fragment of the main class. The payload contains an update engine that requests instructions from a remote Command & Control (C&C) server in the form of a second payload that might change depending on the interests of the attacker and the target device. This second payload is then retrieved, dynamically loaded, and executed in the user device. Initially, the update component used in our first demonstration was programmed to exfiltrate the IMEI of the device. The app was submitted to Google Play in early June 2014 and passed all security controls. At the time of this writing, it is still available in the market[3], although our C&C server has been instructed to always return an innocuous payload and the app was modified to leak nothing from the device.

[2] `https://code.google.com/p/f5-steganography`

[3] `https://play.google.com/store/apps/details?id=es.uc3m.cosec.likeimage`

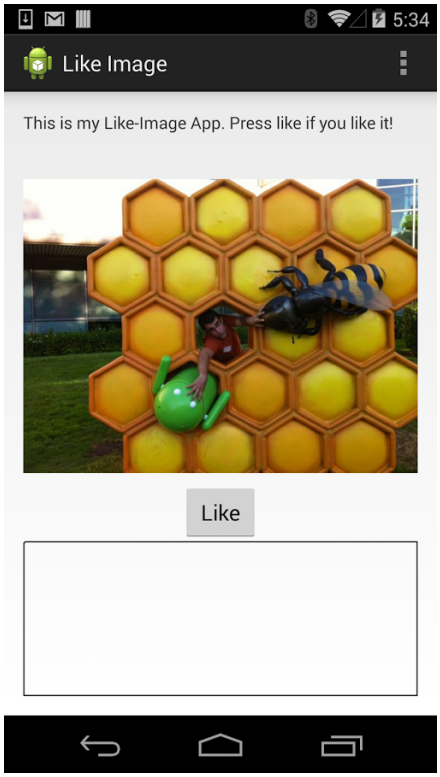


Fig. 1: Our stegomalware app (`LikeImage`).

4 Searching for Stegomalware in the Wild

After the ideas introduced in previous sections, we next describe our efforts so far to find out if malware with steganographic capabilities is already in the wild. Our focus has been on types 0 and I, and we have only searched for apps using image or audio (mp3) stegosystems. With this in mind, we built a detector whose operating principle revolves around three main ideas:

1. Detect the presence of capabilities to execute dynamically loaded code. This is essential to transfer control to any downloaded or extracted payload.
2. Identify assets that are suspect of containing embedded messages (steganalysis).
3. Identify steganographic decoding algorithms in the app code.

In the remaining of this section we describe the experimental setting used in our study and discuss the main results obtained so far.

(a) Original image. (b) Image with embedded payload.

Fig. 2: Asset used by our stegomalware proof-of-concept.

```
package es.uc3m.cosec;
public class Malware{
  String getPayload(){
      String payload;
      ...
      return payload;
  }
}
```

(a) Java code.

```
DEX version '035'
Class #0 -
  Class : 'LMalware;'
  Methods -
    #0 : in LMalware;
    name : 'getPayload'
    type : 'String;'
  source_file: Malware.java
```

(b) DEX dump.

Fig. 3: Code snippet of the malicous payload.

4.1 Experimental Setting

Our study is based on a dataset composed of around 54K apps retrieved from two different sources (see Table 2). On the one hand, we downloaded around 31K apps from Aptoide[4] (AP), a distributed marketplace for Android apps. Contrarily to popular markets such as Google Play, in Aptoide there is not a centralized repository for apps, but each user manages their own "store." On the other hand, we retrieved around 22K Android malware samples from a popular virus repository: VirusShare[5] (VS).

We implemented a detection framework for Android apps in Java and Python. The detector makes use of several open source tools that allows us to extract static information from Android apps and unpackage their resources [5]. It also relies on existing open-source implementations of various steganalytics tools, such as those described in Section 2.3. Finally, we also integrated some parts of the implementation of Anagram [28], an anomaly-based intrusion detection system for application-layer traffic based on n-gram analysis and Bloom filters. As

[4] http://www.aptoide.com/
[5] http://www.virusshare.com/

Table 2: App sources used in our experimentation.

Source	#Apps	Type
Aptoide (AP)	31,935	Presumably goodware
VirusShare (VS)	22,707	Known malware
Total	54,642	

explained in detail later, this is used as the basis for a detector of steganographic code.

4.2 Step 1: Selecting Apps with Payload Execution Capability

The first component in our detection framework identifies apps containing the capability to execute payloads. For instance, Android provides a runtime environment that allows apps to dynamically load libraries. Detecting such capabilities may help to discard those apps that make use of a stegosystem for purposes other than hiding a malicious payload (e.g., to check a watermark).

In our current implementation, we pay special attention to apps with one or more of the following features:

- *Native code*, i.e., apps that contain components using native-code languages such as C and C++.
- *Dynamic code*, i.e. apps that contain functions to dynamically load executable code or libraries.
- *Reflection code*, i.e., apps that attempt to dynamically inspect other fragments of code at runtime.

We assign a "code execution score" S_C to each app $\mathcal{A}$ based on this. Specifically:

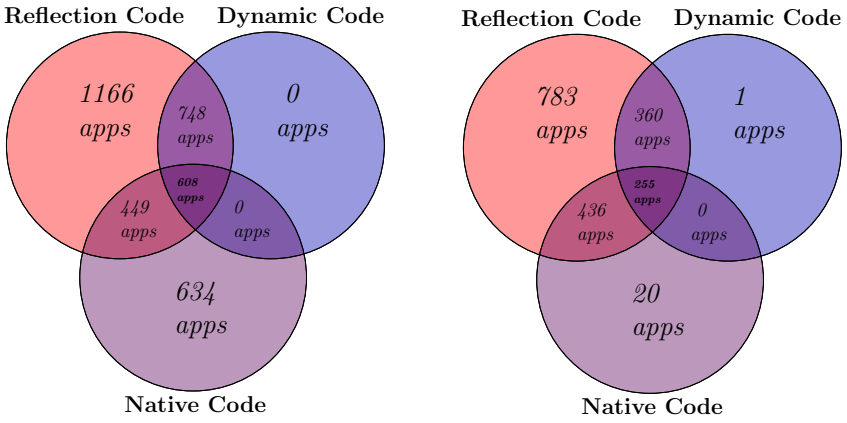
$$S_C(\mathcal{A}) = s_{\text{native}} + s_{\text{dynamic}} + s_{\text{reflection}} \tag{2}$$

where s_x is a factor measuring the risk of having code of type x. We experimentally determined such factors to be $s_{\text{native}} = 0.1$, $s_{\text{dynamic}} = 0.2$, and $s_{\text{reflection}} = 0.5$.

We computed the score defined in Eq. (2) for all apps in our dataset. The results are shown in the Venn diagrams provided in Fig. 4. We discovered that 3,605 apps in the AP dataset (around 11%) contain some form of advanced code. Similarly, our analysis over the VS dataset reported 1,855 samples (around 8%) with advanced code. It can also be observed that reflection code is the most common operation, as it appears in 4,239 and 1,834 apps in AP and VS, respectively.

4.3 Step 2: Flagging Suspicious Assets

Our second contributing factor to the overall detection score is related to the presence of hidden information embedded in the app assets. When analyzing the



(a) Code execution capabilities in Aptoide.

(b) Code execution capabilities in VirusShare.

Fig. 4: Number of apps containing native, dynamic, and reflection code in our dataset.

distribution of potential stegotexts in our dataset, we observed that many apps contain a large number of images. For instance, several apps contained over 5K images each. Additionally, we also found that audio files are less common. In our subsequent analysis, we limited the search for stegotexts to JPEG images and MP3 audio files.

We use `Stegdetect` to determine if a given candidate is likely to contain a hidden payload within any of its image or audio files. `Stegdetect` admits as input a sensitivity threshold ranging between 0.1 and 10 that is used by the underlying distinguishers. The higher this threshold is, the more sensitive the test. Furthermore, the output provides a confidence level in the detection of hidden messages that takes 3 possible values: low confidence (*), medium confidence (**), and high confidence (***).

As in the first step above, we computed a numerical score with each app summarizing the likelihood of having embedded information in its contents. This "steganalysis score" S_H is computed as:

$$S_H(\mathcal{A}) = \max\left\{1, \sum_{c \in R(\mathcal{A})} \frac{\texttt{conf}(c)}{3}\right\} \tag{3}$$

where $R(\mathcal{A})$ is the set of potential stegotexts in the app (i.e., its resources/assets) and $\texttt{conf}(c) \in \{1, 2, 3\}$ is the confidence level returned by `Stegdetect`, or 0 in the case that no distinguisher returns true.

Table 3 summarizes the experimental results obtained and shows the number of images matching each steganographic method for a relatively low sensitivity value (1.0). There is an extremely high number of matches. Specifically, `Stegdetect` reports matches for more than 20K images distributed in approximately 3K apps in the case of AP, and almost 10K images over 2K apps for

Table 3: Number of matches per steganographic tool/method (`JPHide`, `OutGuess`, `JSteg`, `F5`, `appended`, `camouflage`, and `alpha-channel`) and confidence (*, **, and ***) for all samples retrieved from Aptoide (AP) and VirusShare (VS).

	JPHide			**OutGuess**			**JSteg**			**F5**			**Other**		
	(*)	(**)	(***)	(*)	(**)	(***)	(*)	(**)	(***)	(*)	(**)	(***)	appended	camouflage	alpha-channel
AP	9,774	2,219	8,421	517	441	2,203	10	5	0	0	0	3	257	1	5
	20,414			3,161			15			3			263		
	23,804 matches in 3,364 apps														
VS	3,924	3,924	2,570	137	143	1,260	6	0	0	0	0	0	213	0	0
	10,418			1,540			6			0			213		
	12,177 matches in 2,009 apps														

VS. In terms of embedding algorithms, `JPHide` seems to be the most popular steganographic tool used with about 85% the matches for both AP and VS apps. These results will be further discussed later when analyzing individual apps.

4.4 Step 3: Identifying Steganographic Code

A common factor between Type 0 and Type I stegomalware is the presence of a steganographic decoding algorithm SD in the app code required to extract the malicious payload from the stegotext (asset). Determining if a piece of compiled code includes some unknown function SD is not straightforward. We approached this problem statistically using N-gram analysis as follows:

1. For each candidate steganographic decoding algorithm SD that we wish to detect, we create a simple app with just one class that uses SD.
2. We extract from the compiled app all N-grams. To do this, the app is treated as a stream of bytes and a sliding window of length N is passed through it. To efficiently store all N-grams, we relied on the Bloom filter implementation used in Anagram [28] for a content-based anomaly detector.
3. In detection mode, we extract all N-grams from a candidate app and check how many of them are found in the content filter associated with SD. The more the number of hits, the more likely the app contains an implementation of SD identical to that used to generate the content filter.

We generated content filters for various Java open-source steganographic tools, including all stegosystems enumerated in Section 2. We experimented with different values of N and noticed that for $N < 10$ it generates too many false positives, while too high values (i.e., $N > 50$) results in a very ineffective detector. Thus, for each candidate algorithm SD we created 5 different models, one for each $N = \{10, 20, 30, 40, 50\}$.

As with the two previous factors, we associate with each app a score quantifying the likelihood of the app containing one of the steganographic decoding algorithms SD modeled in our content filters. The score is computed in two steps. First, for each algorithm SD and each value of N we compute a normality score by counting the fraction of N-grams of the app $\mathcal{A}$ that are found in the content

filter:

$$S_S(\mathcal{A}, \mathsf{SD}, N) = \frac{\#Hits}{|\mathcal{A}| - N + 1} \tag{4}$$

The score $S_A(\mathcal{A}, \mathsf{SD})$ for algorithm SD is then computed as the average score for the five values of N. Finally, the overall score for the app $S_S(\mathcal{A})$ is just the average score for all algorithms SD. In our experimentation, we determined that a good practice is filtering out those algorithms whose score $S_S(\mathcal{A}, \mathsf{SD})$ is lower than a given threshold (around 0.6 for our datasets). This is the approach followed in the results reported next.

Fig. 5 shows the score distribution computed for those apps that had one or more of the code execution features described in Section 4.2. The score is extremely low for most of them, suggesting that they do not contain traces of steganographic algorithms—at least, not the ones for which we have a content filter. There are, however, a reduced number of apps that seem to contain steganographic-like operations, with scores ranging from 0.2 to almost 0.6. The majority of these apps come from the AP dataset.

Note that we have not considered the scenario where an attacker uses multiple SD algorithms at the same time. Although our implementation reports the average anomaly score per SD, we could easily extend it to evaluate all SD together.

4.5 Putting It All Together

The three separate scores introduced above can be combined into a single value to measure the likelihood of an app containing stegomalware. We propose using

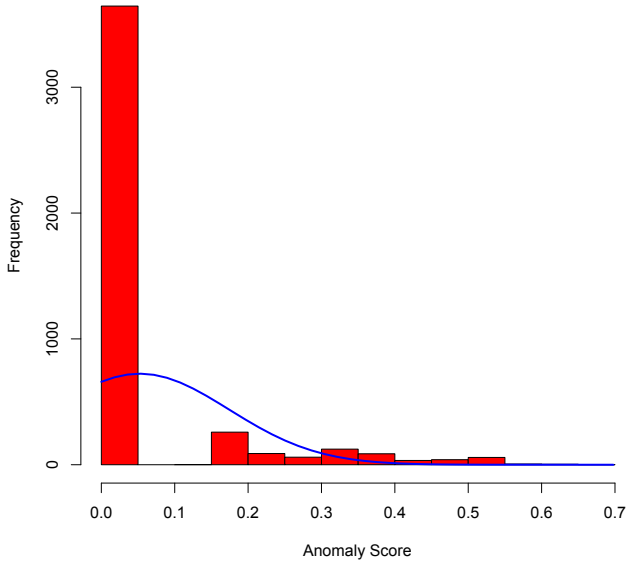


Fig. 5: Distribution of $S_S(\mathcal{A})$.

Table 4: Top 10 scores for specimens from Aptoide and VirusShare. The score of our stegomalware PoC app `Likeimage` is shown as a reference.

Rank	Aptoide	VirusShare
1	0.767	0.746
2	0.734	0.731
3	0.725	0.584
4	0.724	0.068
5	0.701	0.068
6	0.699	0.068
7	0.699	0.067
8	0.697	0.067
9	0.695	0.067
10	0.695	0.066
Our PoC `Likeimage`:	0.738	

the following score:

$$S(\mathcal{A}) = S_S(\mathcal{A})^{1-S_H(\mathcal{A})\cdot S_C(\mathcal{A})} \tag{5}$$

For instance, an app containing dynamic and reflection code, for which the SD detection score is 0.6, and with one picture with hidden content with confidence 100% (***) is assigned a score of $0.6^{1-(1.0\cdot 0.7)} = 0.858$.

The rationale behind expression (5) is to have a power-law score where the basis is determined by the likelihood of detecting a steganographic decoding algorithm in the app. If one is reasonably sure of the fact that the app has steganographic capabilities, then the base score is raised to a factor that considers the existence of code execution capabilities and the output of steganalysis over the app's assets. In doing so, we pursue to reduce the effects of the high false positive ratio of current steganalysis techniques.

4.6 Main Findings

Table 4 shows the top 10 specimens with the highest score in the AP and VS datasets. In VS there is only three samples with a score higher than 0.5, whereas in Aptoide more than 200 samples present such higher scores. For a reference value, our stegomalware demo app `LikeImage` has an associated score of $0.64^{1-(1\cdot 0.7)} = 0.738$.

We carried out some manual inspection of the top scored apps from both Aptoide and VirusShare. We next summarize our main findings:

- ***Most of the inspected apps are definitely not stegomalware as defined in this paper, but many of them behave very much like it.*** We found various cases of apps manipulating images through operations very similar to those used by some LSB encoding algorithms. For instance, several apps use some algorithms contained in the Apache Commons Imaging

```
VS sample a8869e0ec4fb1acf12ae01d5294c0c2b.apk
    91138622720e0cf3b119c1c50846f21fbf09aae7.jpg - jphide(*)
    6391e903918fa0ecc9951f32249759ee3d6ddb04.jpg - jphide(*)
    f76575600c3387444c079942530fd9f9d62aa08b.jpg - jphide(*)
    e8112b2ac65c10384e23dff0b0119313b07e8967.jpg - jphide(**)
    ae8267310a55b3192befac4041a98226cffc17b5.jpg - jphide(**)
    5d212aa85edf8db1664c98520b23dd54564e7463.jpg - jphide(***)
    9304c888d43f87943d766a12d01b0ef41ad53ac5.jpg - jphide(***)
    5af4d7ea15ce36d3a21d75ff3bf33a87e850b1a1.jpg - jphide(***)
    f35ea0096b63f624f13782ce8644ebf81b4ca352.jpg - jphide(***)
    90cebeec08fa513d9a1b0ce03f6d55fbb2fbd918.jpg - jphide(***)
    dc854fda81cb39db12584466d2160924ab183022.jpg - jphide(***)
    834344afa40f4bfb7391e165014f78f0f736187a.jpg - jphide(***)
    78701455b319ebc4a4ff8aa18026cffc1f1716c7.jpg - jphide(***)
    ...
```

Fig. 6: Fragment of the `Stegdetect` report on various images extracted from a VS sample. The app produced over 70 hits.

library[6], including Huffman coding for constructing minimum-redundancy codes, which is used by some steganographic tools.

– ***A huge amount of apps contain images that clearly have embedded information.***

For example, Fig. 6 shows a fragment of the report on a VS sample with more than 70 (***) hits. However, to the best of our understanding, such hidden messages are never extracted during the app execution. We did not try to break the password by brute force and recover the embedded messages, but the headers used by the encoding algorithms are recognizable. Interestingly, many of these apps use cryptographic functions either to obfuscate payloads piggybacked together with the app (in the case of VS apps), or else as part of the legitimate function of the app (in some AP apps).

Such a large amount of images with hidden content suggest two different conclusions. Firstly, many apps might contain stegotexts for purposes other than stegomalware as defined in this paper. Some form of watermarking or other copyright protection techniques is a natural explanation, but they might have other uses. Secondly, even though Provos and Honeyman [19] report a very low percentage of false positives for `Stegdetect`, a recent study by Khalind et al. [14] suggests that it depends on the chosen sensitivity and that it can be quite high in some cases. Thus, it may be the case that a fraction of those images do not actually have any embedded information.

– ***We found many cases of apps that use naïve methods to hide their malicious components.***

For instance, the sample with identifier

`f0f65bd7287cf 83bfabcd22cbf6a0c8c`

[6] `http://commons.apache.org/proper/commons-imaging/`

from VS simply stores the payload in a file called *data.png*. At runtime it uses several methods, including one with the suspicious name:

```
cn.bighead.utils.Encoding.covert2Url(covert)
```

to extract the required components. Fig. 7 shows additional examples of suspicious operations used to conceal malicious information.

5 Conclusions and Future Work

In this paper, we have introduced and discussed the notion of stegomalware, a class of malware attacks using information hiding techniques to evade detection and hinder the task of security analysts. We have argued that this is a far more powerful capability than traditional obfuscation techniques currently observed in most smartphonne malware samples, such as disguising the malicious payload as an image or audio file by simply faking the file extension, or putting it at the end of another asset. We have introduced different architectural choices for smartphone stegomalware and demonstrated its viability with a proof-of-concept app that has remained undetected in the Google Play market for nearly 4 months so far.

In an attempt to check whether something similar to our notion of stegomalware is actually being used, we have proposed a detection framework that combines evidences gathered from the use of code execution capabilities, the presence of hidden messages detected by steganalysis, and the identification of steganographic decoding algorithms in an app's code. We have applied our detector to a dataset of around 55K apps comprising both known malware and apps gathered from an alternative market.

Unfortunately, the preliminary results of our search for stegomalware in the wild are not conclusive. However, our study so far has produced a few interesting conclusions. For example, we have found that a substantial amount of apps do

```
public long a(byte[] param){
  return
  (0xFF & param[3]) +
  ((0xFF & param[4]) << 16) +
  ((0xFF & param[2]) << 40) +
  ((0xFF & param[0]) << 48) +
  (param[1] << 56);
}
```

(a) Sample `224058dbe82d71248cf93c5` transforms an array of bytes and returns the hidden value.

```
private static String a(){
  return new StringBuilder()
          .append(i.a(200))
          .append(i.a(194))
          .append(i.a(216))
          .append(i.a(92))
          .toString();
}
```

(b) Sample `a8869e0ec4017d5294c0c2b` plays with the way strings are declared in order to hide them.

Fig. 7: Examples of methods found in various apps to hide malicious components.

have assets with hidden information. Furthermore, many goodware apps such as games, productivity, and e-health tools incorporate steganographic decoding algorithms. We do not claim that such apps are stegomalware as defined in this paper, since there are many legitimate uses of steganography (e.g., intellectual property protection). Nevertheless, we believe this issue deserves more attention. In particular, *we recommend market operators to include steganalysis and other forms of stegomalware detection among their analysis techniques.*

Our work can be extended in a number of ways. For instance, the detection framework implemented so far only includes image and audio steganography. There are stegosystems for many other digital objects that are often found among an app's resources, such as XML and HTML documents, databases, and other multimedia files. We will incorporate appropriate distinguishers and content filters to our prototype to check such assets.

Acknowledgements

We are very grateful to the anonymous reviewers for constructive feedback and insightful suggestions that helped to improve the quality of the original manuscript. This work was supported by the MINECO grant TIN2013-46469-R (SPINY: Security and Privacy in the Internet of You).

References

1. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K.: Drebin: Effective and explainable detection of android malware in your pocket. In: Proc. of Network and Distributed System Security Symposium (NDSS) (February 2014)
2. Bastien, F.: Sss – simple steganalysis suite (Visited 2014), `https://code.google.com/p/simple-steganalysis-suite/`
3. Cachin, C.: Digital steganography. In: Encyclopedia of Crypto. and Security. Springer (2005)
4. Cheddad, A., Condell, J., Curran, K., Mc Kevitt, P.: Digital image steganography: Survey and analysis of current methods. Signal processing 90(3), 727–752 (2010)
5. Desnos, A., et al.: Androguard: Reverse engineering, malware and goodware analysis of android applications (Visited December 2013), `https://code.google.com/p/androguard`
6. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comp. Surv. 44(2), 1–42 (Mar 2012)
7. Farid, H., Siwei, L.: Detecting hidden messages using higher-order statistics and support vector machines. In: LNCS, vol. 2578. pp. 340–354. Springer-Verlag (2003)
8. Forczmanski, P., Wegrzyn, M.: Open virtual steganographic laboratory. In: International Conference on Advanced Computer Systems (ACS-AISBIS) (2009), `http://vsl.sourceforge.net/`
9. Fridrich, J.: Feature-based steganalysis for jpeg images and its implications for future design of steganographic schemes. In: Info. Hiding. pp. 67–81. Springer (2005)
10. Fridrich, J., Goljan, M., Hogea, D.: New methodology for breaking steganographic techniques for jpegs. In: Electronic Imaging 2003. pp. 143–155. International Society for Optics and Photonics (2003)

11. Gao, J., Bai, X., Tsai, W.T., Uehara, T.: Mobile application testing: A tutorial. Computer 47(2), 46–55 (Feb 2014)
12. Huang, H., Zhu, S., Liu, P., Wu, D.: A framework for evaluating mobile app repackaging detection algorithms. In: Trust and Trustworthy Comput., pp. 169–186 (2013)
13. Johnson, N.F., Jajodia, S.: Exploring steganography: Seeing the unseen. Computer 31(2), 26–34 (1998)
14. Khalind, O.S., Hernandez-Castro, J.C., Aziz, B.: A study on the false positive rate of stegdetect. Digital Investigation 9(3), 235–245 (2013)
15. Oberheide, J., Miller, C.: Dissecting the android bouncer. In: SummerCon (2012)
16. O'Kane, P., Sezer, S., McLaughlin, K.: Obfuscation: The hidden malware. IEEE Security & Privacy 9(5), 41–47 (2011)
17. Petitcolas, F.A., Anderson, R.J., Kuhn, M.G.: Information hiding-a survey. Proceedings of the IEEE 87(7), 1062–1078 (1999)
18. Pfitzmann, B.: Information hiding terminology. In: Information Hiding, First International Workshop. LNCS, vol. 1174, pp. 347–350. Springer (1996)
19. Provos, N., Honeyman, P.: Hide and seek: an introduction to steganography. Security Privacy, IEEE 1(3), 32–44 (May 2003)
20. Provos, N., Honeyman, P.: Detecting steganographic content on the internet. Tech. rep., Center for Information Technology Integration University of Michigan (2001)
21. Rastogi, V., Chen, Y., Enck, W.: Appsplayground: automatic security analysis of smartphone applications. In: Proceedings of the third ACM conference on Data and application security and privacy. pp. 209–220. CODASPY '13, ACM, New York, NY, USA (2013)
22. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. pp. 329–334. ASIA CCS '13, ACM, New York, NY, USA (2013)
23. Shabtai, A., Tenenboim-Chekina, L., Mimran, D., Rokach, L., Shapira, B., Elovici, Y.: Mobile malware detection through analysis of deviations in application network behavior. Computers & Security (2014)
24. Suarez-Tangil, G., Tapiador, J.E., Lombardi, F., Pietro, R.D.: Thwarting obfuscated malware via differential fault analysis. IEEE Computer 47(6), 24–31 (2014)
25. Suarez-Tangil, G., Tapiador, J.E., Peris, P., Ribagorda, A.: Evolution, detection and analysis of malware for smart devices. IEEE Communications Surveys & Tutorials 16(2), 961–987 (May 2014)
26. Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Blasco, J.: Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. Expert Systems with Applications 41(1), 1104–1117 (2014)
27. Upham, D.: Jsteg (1997), `http://www.tiac.net/users/korejwa/jsteg.htm`
28. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A content anomaly detector resistant to mimicry attack. In: Advances in Intrusion Detection. pp. 226–248 (2006)
29. Westfeld, A.: F5–a steganographic algorithm. In: Info. hiding. pp. 289–302 (2001)
30. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: IEEE Symposium on Security and Privacy. pp. 95–109 (2012)